

# Python for Data Processing and Climate Analysis

Hamid Oloso & Jules Kouatchou  
November 2012

# Source

Part of this tutorial is based on the materials taken from:

**Python Scripting for Computational Science**, by Hans Petter Langtangen

<http://heim.ifi.uio.no/~hpl/scripting/all-nosplit>

**A Hands-On Introduction to Using Python in the Atmospheric and Oceanic Sciences**, by Johnny Wei-Bing Lin

<http://www.johnny-lin.com/pyintro/>

# Obtaining the Material

Materials for this training are in the file *pythonTrainingGISS.tar.gz* that is available at:

**[modelingguru.nasa.gov/docs/DOC-2292](http://modelingguru.nasa.gov/docs/DOC-2292)**

Do the following:

```
tar xzvf pythonTrainingGISS.tar.gz
```

You will get the directory *pythonTraingGISS* that contains:

*Slides/*

*Examples/*

# Settings on discover/dali

We installed a Python distribution. To use it, you need to load the modules:

```
module load other/comp/gcc-4.5
```

```
module load other/SIVO-PyD/spd_1.6.0_GISS
```

# SIVO-PyD

- Collection of Python packages for scientific computing and visualization
- All the packages are accessible within the Python framework
- Self-contained distribution that mimics the commercial Enthought one.

# Settings on your Local Mac

Go to the following Modeling Guru page:

<https://modelingguru.nasa.gov/docs/DOC-1847>

and follow the instructions to install Python, Numpy, SciPy, Matplotlib and Basemap.

# Available Python Related Resources

**SIVO-PyD: A Python Distribution for Scientific Computing and Data Visualization**

<https://modelingguru.nasa.gov/docs/DOC-2109>

**Installing Python, Numpy, Matplotlib and Basemap on the Mac**

<https://modelingguru.nasa.gov/docs/DOC-1847>

**Comparing Python, Numpy, Matlab, Fortran, etc.**

<https://modelingguru.nasa.gov/docs/DOC-1762>

**Various Scripts**

PoDS: <https://modelingguru.nasa.gov/docs/DOC-1582>

# Cores/Node: <https://modelingguru.nasa.gov/docs/DOC-1765>

Utilities: <https://modelingguru.nasa.gov/docs/DOC-1816>

# What Will be Covered?

- **Python**
- **Numpy**
- **SciPy**
- **Matplotlib**
- **netCDF4**
- **EOFs**

# Python

- **Python expressions**
- **Basic I/O**
- **If statements, for loops, while loops**
- **Creating modules**
- **Lists**
- **Classes in Python**

# What is Python?

**Python** is an elegant and robust programming language that combines the power and flexibility of traditional compiled languages with the ease-of-use of simpler scripting and interpreted languages.

# What is Python?

- High level
- Interpreted
- Scalable
- Extensible
- Portable
- Easy to learn, read and maintain
- Robust
- Object oriented
- Versatile

# Why Python?

- Free & Open source
- Built-in run-time checks
- Nested, heterogeneous data structures
- OO programming
- Support for efficient numerical computing
- Good memory management
- Can be integrated with C, C++, Fortran and Java
- Easier to create stand-alone applications on any platform

# Scientific Hello World

- Provide a number to the script
- Print “Hello World” and the sine value of the number

To run the script, type:

```
./helloWorld.py 3.14
```

## Purpose of the Script

- Read a command line argument
- Call a math (sine) function
- Work with variables
- Print text and numbers

# The Code

```
#!/usr/bin/env python

#-----
# Load modules
#-----

import sys
import math

#-----
# Extract the 1st command-line argument
#-----

r = float(sys.argv[1])

s = math.sin(r)
print "Hello, World! sin(" + str(r) + ")=" + str(s)
```

# Header

- **Explicit path to the interpreter:**

*#!/usr/bin/python*

or perhaps your own Python interpreter:

*#!/usr/local/other/Python-2.5.4/bin/python*

- **Using env to find the first Python interpreter in the path:**

*#!/usr/bin/env python*

# Importing Python Modules

The standard way of loading a module is:

```
import scipy
```

We can also use:

```
from scipy import *
```

We may choose to load a “sub-module” of the main one:

```
import scipy.optimize  
from scipy.optimize import *
```

We can choose to retrieve a specific function of a module:

```
from scipy.optimize import fsolve
```

You can even rename a module:

```
import scipy as sp
```

```
dir(scipy)            help(scipy)
```

# Alternative Print Statements

- String concatenation:

```
print "Hello, World! sin(" + str(r) + ")=" + str(s)
```

- C printf-like statement:

```
print "Hello, World! sin(%g)=%g" % (r,s)
```

- Variable interpolation:

```
print "Hello, World! sin(%(r)g)=%(s)g" % vars()
```

# Printf format strings

%d : integer

%5d : integer in a field of width 5 chars

%-5d : integer in a field of width 5 chars, but adjusted to the left

%05d : integer in a field of width 5 chars, padded with zeroes from the left

%g : float variable in %f or %g notation

%e : float variable in scientific notation

%11.3e : float variable in scientific notation, with 3 decimals, field of width 11 chars

%5.1f : float variable in fixed decimal notation, with one decimal, field of width 5 chars

%.3f : float variable in fixed decimal form, with three decimals, field of min. width

%s : string

%-20s : string in a field of width 20 chars, and adjusted to the left

# Python Types

- Numbers: float, complex, int (+ bool)
- Sequences: list, tuple, str, NumPy arrays
- Mappings: dict (dictionary/hash)
- Instances: user-defined class
- Callables: functions, callable instances

# Numerical Expressions

- Python distinguishes between strings and numbers:

```
b = 1.2                # b is a number  
b = '1.2'            # b is a string  
a = 0.5 * b          # illegal: b is NOT converted to float  
a = 0.5 * float(b)  # this works
```

- All Python objects are compared with

```
==  !=  <  >  <=  >=
```

# Boolean Expressions

- **bool** is **True** or **False**
- Can mix **bool** with **int** **0** (false) or **1** (true)
- Boolean tests:

```
a = ""; a = []; a = (); a = {};      # empty structures
```

```
a = 0; a = 0.0
```

```
if a:                                # false
```

```
if not a:                             # true
```

other values of **a**: **if a** is true

# Strings

- **Single- and double-quoted strings work in the same way**

*s1 = "some string with a number %g" % r*

*s2 = 'some string with a number %g' % r # = s1*

- **Triple-quoted strings can be multi line with embedded newlines:**

*text = """*

*large portions of a text can be conveniently placed*

*inside triple-quoted strings (newlines are preserved)"""*

- **Raw strings, where backslash is backslash:**

*s3 = r'(\|s+|. \|d+|)'*

*# with ordinary string (must quote backslash):*

*s3 = '\\(\\|s+|\\. \|d+|\\)'*

# Variables and Data Types

Type	Range	To Define	To Convert
float	numbers	<code>x=1.0</code>	<code>z=float(x)</code>
integer	numbers	<code>x=1</code>	<code>z=int(x)</code>
complex	complex numbers	<code>x=1+3j</code>	<code>z=complex(a,b)</code>
string	text strings	<code>x= 'test'</code>	<code>z=str(x)</code>
boolean	True or False	<code>x=True</code>	<code>z=bool(x)</code>

# If Statements

```
if <conditions>:  
    <statements>  
elif <conditions>:  
    <statements>  
else:  
    <statements>
```

```
x = 10  
if x > 0:  
    print 1  
elif x == 0:  
    print 0  
else:  
    print -1
```

# For Loops

For loops iterate over a sequence of objects

```
for <loop_var> in <sequence>:  
    <statements>
```

```
for i in range(5):  
    print i,
```

```
for i in 'abcde':  
    print i,
```

```
l=['dogs', 'cats', 'bears']  
accum = ""  
for item in l:  
    accum = accum + item  
    accum = accum + ' '
```

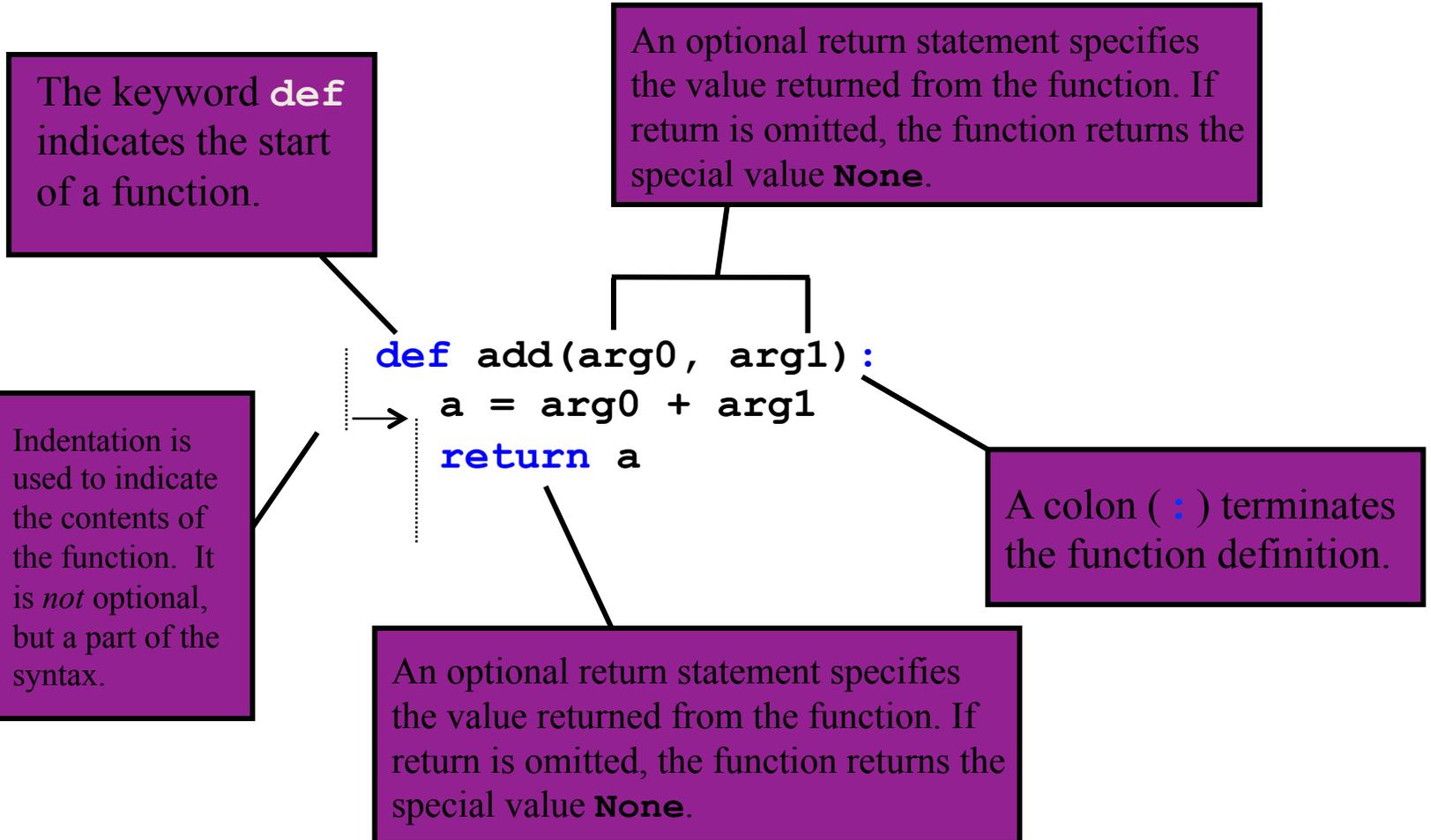
# While Loop

```
while <condition>:  
    <statements>
```

```
lst = range(3)  
while lst:  
    print lst  
    lst = lst[1:]
```

```
i = 0  
while 1:  
    if i < 3:  
        print i,  
    else:  
        break  
    i = i + 1
```

# Functions



# Reading/Writing Data Files

## Tasks:

- Read  $(x,y)$  data from a two-column file
- Transform  $y$  values to  $f(y)$
- Write  $(x,f(y))$  to a new file

## What to learn:

- How to open, read, write and close files
- How to write and call a function
- How to work with arrays (lists)

# Reading Input/Output Filenames

- Usage:

*./readInputOutputFiles1.py infilename outfile*

- Read the two command-line arguments: input and output filenames

*infile* = *sys.argv[1]*

*outfile* = *sys.argv[2]*

- Command-line arguments are in **sys.argv[1:]**
- **sys.argv[0]** is the name of the script

# Exception Handling

- What if the user fails to provide two command-line arguments?
- Python aborts execution with an informative error message
- Manual handling of errors:

*try:*

```
infilename = sys.argv[1]
```

```
outfilename = sys.argv[2]
```

*except:*

```
# try block failed, we miss two command-line arguments
```

```
print 'Usage:', sys.argv[0], 'infile outfile'
```

```
sys.exit(1)
```

This is the common way of dealing with errors in Python, called *exception handling*

# Open File and Read Line by Line

- Open files:

```
ifile = open( infilename, 'r') # r for reading
```

```
ofile = open(outfilename, 'w') # w for writing
```

```
afile = open(appfilename, 'a') # a for appending
```

- Read line by line:

```
for line in ifile:
```

```
    # process line
```

- Observe: blocks are indented; no braces!

# Defining a Function

```
import math
def myfunc(y):
    if y >= 0.0:
        return y**5*math.exp(-y)
    else:
        return 0.0
```

```
# alternative way of calling module functions
# (gives more math-like syntax in this example):
```

```
from math import *
def myfunc(y):
    if y >= 0.0:
        return y**5*exp(-y)
    else:
        return 0.0
```

# Data Transformation Loop

- Input file format: two columns with numbers

*0.1 1.4397*

*0.2 4.325*

*0.5 9.0*

- Read (x,y), transform y, write (x,f(y)):

*for line in ifile:*

*pair = line.split()*

*x = float(pair[0])*

*y = float(pair[1])*

*fy = myfunc(y) # transform y value*

*ofile.write('%g %12.5e\n' % (x,fy))*

# Alternative File Reading

- This construction is more flexible and traditional in Python:

```
while 1:  
    line = ifile.readline() # read a line  
    if not line: break  
    # process line
```

i.e., an 'infinite' loop with the termination criterion inside the loop

# Loading Data into Lists

- Read input file into list of lines:

```
lines = ifile.readlines()
```

- Now the 1st line is `lines[0]`, the 2nd is `lines[1]`, etc.
- Store x and y data in lists:

```
# go through each line, split line into x and y columns
```

```
x = []
```

```
y = []
```

```
for line in lines:
```

```
    xval, yval = line.split()
```

```
    x.append(float(xval))
```

```
    y.append(float(yval))
```

# Loop over List Entries

- For-loop in Python:

```
for i in range(start,stop,inc):
```

```
...
```

```
for j in range(stop):
```

```
...
```

Generates

```
i = start, start+inc, start+2*inc, ..., stop-1
```

```
j = 0, 1, 2, ..., stop-1
```

- Loop over (x,y) values:

```
ofile = open(outfilename, 'w') # open for writing
```

```
for i in range(len(x)):
```

```
    fy = myfunc(y[i]) # transform y value
```

```
    ofile.write('%g %12.5e\n' % (x[i], fy))
```

```
ofile.close()
```

# Error Checking

- Easy to introduce intricate bugs
  - no declaration of variables
  - functions can "eat anything"
- No, extensive consistency checks at run time replace the need for strong typing and compile-time checks
- Example: sending a string to the sine function, `math.sin('t')`, triggers a run-time error (type incompatibility)
- Example: try to open a non-existing file

*./readInputOutputFiles2.py qq someoutfile*

*Traceback (most recent call last):*

*File "./readInputOutputFiles2.py", line 12, in ?*

*ifile = open( infile, 'r')*

*IOError:[Errno 2] No such file or directory:'qq'*

*./readInputOutputFiles2.py infile outfile*

# Computing with Arrays

- $x$  and  $y$  in *readInputOutputFiles2.py* are *lists*
- We can compute with lists element by element (as shown)
- However: using Numerical Python (NumPy) *arrays instead of lists is much more efficient and convenient*
- Numerical Python is an extension of Python: a new fixed-size array type and lots of functions operating on such arrays

# Interactive Computing with **ipython**

- Allows to run commands interactively
- Prompts you to executes any valid Python statement
- Executes Python scripts: “run myFile.py args”
- Points to any error
- Provides help functionality: “help numpy.random”
- Up- and down-arrows: go through command history
- The underscore variable holds the last output

```
In [6]:y
```

```
Out[6]:0.93203908596722629
```

```
In [7]:_ + 1
```

```
Out[7]:1.93203908596722629
```

# Ipython – TAB Completion

- IPython supports TAB completion: write a part of a command or name (variable, function, module), hit the TAB key, and IPython will complete the word or show different alternatives:

```
In [1]: import math
```

```
In [2]: math.<TABKEY>
```

```
math.__class__
```

```
math.__str__
```

```
math.frexp
```

```
math.__delattr__
```

```
math.acos
```

```
math.hypot
```

```
math.__dict__
```

```
math.asin
```

```
math.ldexp...
```

or

```
In [2]: my_variable_with_a_very_long_name = True
```

```
In [3]: my<TABKEY>
```

```
In [3]: my_variable_with_a_very_long_name
```

You can increase your typing speed with TAB completion!

# IPython and the Python Debugger

- Scripts can be run from IPython:

```
In [1]:run scriptfile arg1 arg2 ...
```

e.g.,

```
In [1]:run readInputOutputfiles2.py .datatrans_infile tmp1
```

- IPython is integrated with Python's **pdb** debugger
- **pdb** can be automatically invoked when an exception occurs:

```
In [29]:%pdb on # invoke pdb automatically
```

```
In [30]:run readInputOutputfiles2.py infile tmp2
```

# The os Module

- Python has a rich cross-platform operating system (OS) interface
- Skip Unix- or DOS-specific commands; do all OS operations in Python!
- The os module provides dozens of functions for interacting with the operating system

```
import os
```

```
dir(os)
```

```
help(os)
```

# Some os Functions

```
curDir = os.getcwd()           # Return the current working directory
os.chdir('targetDir')         # Change directory
os.remove("file")             # Remove file
os.rename("oldName", "newName") # Rename file
os.listdir("myDir")           # Provide a list of files in myDir
os.removedirs("myDir")        # Remove all empty directories above myDir
os.system("command")          # Execute the command
```

# Creating a Subdirectory

Safe creation of a subdirectory:

```
dir = case                # subdirectory name
import os, shutil
if os.path.isdir(dir):    # does dir exist?
    shutil.rmtree(dir)    # yes, remove old files
os.mkdir(dir)             # make dir directory
os.chdir(dir)            # move to dir
```

# Creating Your Own Module (1)

```
#!/usr/bin/env python
# fileName: mymodule.py

def sayhi(name):
    print 'Hi from %s: this is mymodule speaking.' %(name)

version = '0.1'
# End of mymodule.py
```

- Remember that the module should be placed in the same directory as the program that we import it in, or
- the module should be in one of the directories listed in **sys.path**.

# Creating Your Own Module (2)

```
#!/usr/bin/env python
# filename: mymodule_demo.py

import mymodule

mymodule.sayhi('Jules')
print 'Version', mymodule.version
# End of mymodule_demo.py
```

If you run the above script:

```
./mymodule_demo.py
```

You will get:

```
Hi from Jules: this is mymodule speaking.
Version 0.1
```

# Executing Modules as Scripts (1)

- We want to execute the code in the module as it was imported
- We need to add the following at the end of the module:

```
if __name__ == "__main__":  
    # section of the module to be executed
```

# Executing Modules as Scripts (2)

```
#!/usr/bin/env python
# fileName: mymodule.py

def sayhi(name):
    print "Hi from %s: this is mymodule speaking." %(name)
version = '1.0'

if __name__ == "__main__":
    import sys
    try:
        sayhi(str(sys.argv[1]))
    except:
        print 'Usage: %s string' % sys.argv[0]
        sys.exit(1)
    print 'Version', version
# End of mymodule.py
```

Only need to type: `./mymodule.py Jules`

# Setting List Elements

- Initializing a list:

```
arglist = [myarg1, 'displacement', "tmp.ps"]
```

- Or with indices (if there are already two list elements):

```
arglist[0] = myarg1
```

```
arglist[1] = 'displacement'
```

- Create list of specified length:

```
n = 100
```

```
mylist = [0.0]*n
```

- Adding list elements:

```
arglist = [] # start with empty list
```

```
arglist.append(myarg1)
```

```
arglist.append('displacement')
```

# Getting List Elements

- Extract elements from a list:  
`filename, plottitle, psfile = arglist`  
`(filename, plottitle, psfile) = arglist`  
`[filename, plottitle, psfile] = arglist`
- Or with indices:  
`filename = arglist[0]`  
`plottitle = arglist[1]`

# Traversing Lists

- For each item in a list:  

```
for entry in arglist:  
    print 'entry is', entry
```
- For-loop-like traversal:  

```
start = 0  
stop = len(arglist)  
step = 1  
for index in range(start, stop, step):  
    print 'arglist[%d]=%s' % (index,arglist[index])
```
- Visiting items in reverse order:  

```
mylist.reverse()          # reverse order  
for item in mylist:  
    # do something...
```

# List Comprehensions

- Compact syntax for manipulating all elements of a list:

```
y = [ float(yi) for yi in line.split() ]      # call function float  
x = [ a+i*h for i in range(n+1) ]          # execute expression
```

(called list comprehension)

- Written out:

```
y = []  
for yi in line.split():  
    y.append(float(yi))  
etc.
```

# Map Function

- map is an alternative to list comprehension:

```
y = map(float, line.split())
```

```
y = map(lambda i: a+i*h, range(n+1))
```

- map is faster than list comprehension but not as easy to read

# Typical List Operations

```
d = []           # declare empty list
d.append(1.2)    # add a number 1.2
d.append('a')   # add a text
d[0] = 1.3      # change an item
del d[1]        # delete an item
len(d)         # length of list
d.count(x)      # count the number of times x occurs
d.index(x)      # return the index of the first occurrence of x
d.remove(x)     # delete the first occurrence of x
d.reverse      # reverse the order of elements in the list
```

# Nested Lists

- Lists can be nested and heterogeneous
- List of string, number, list and dictionary:

```
>>> mylist = ['t2.ps', 1.45, ['t2.gif', 't2.png'],\
               { 'factor' : 1.0, 'c' : 0.9} ]
>>> mylist[3]
{'c': 0.90000000000000002, 'factor': 1.0}
>>> mylist[3]['factor']
1.0
>>> print mylist
['t2.ps', 1.45, ['t2.gif', 't2.png'],
 {'c': 0.90000000000000002, 'factor': 1.0}]
```
- Note: print prints all basic Python data structures in a nice format

# Sorting a List

- In-place sort:  
**mylist.sort()**

modifies **mylist**!

```
>>> print mylist
[1.4, 8.2, 77, 10]
>>> mylist.sort()
>>> print mylist
[1.4, 8.2, 10, 77]
```

- Strings and numbers are sorted as expected

# Defining the Comparison Criterion

# ignore case when sorting:

```
def ignorecase_sort(s1, s2):  
    s1 = s1.lower()  
    s2 = s2.lower()  
    if s1 < s2: return -1  
    elif s1 == s2: return 0  
    else: return 1
```

# or a quicker variant, using Python's built-in cmp function:

```
def ignorecase_sort(s1, s2):  
    s1 = s1.lower(); s2 = s2.lower()  
    return cmp(s1,s2)
```

# usage:

```
mywords.sort(ignorecase_sort)
```

# Indexing

```
# list
```

```
# indices: 0 1 2 3 4
```

```
>>> l = [10,11,12,13,14]
```

```
>>> l[0]
```

```
10
```

```
# negative indices count backward from the end of the list
```

```
# indices: -5 -4 -3 -2 -1
```

```
>>> l = [10,11,12,13,14]
```

```
>>> l[-1]
```

```
14
```

```
>>> l[-2]
```

```
13
```

# Slicing

**var[lower:upper]**

Slices extract a portion of a sequence by specifying a lower and upper bound. The extracted elements start at lower and go up to, but do not include, the upper element. Mathematically the range is [lower,upper).

```
>>> l = [10,11,12,13,14]
>>> l[1:3]
[11,12]
>>> l[1,-2]
[11,12]
>>> l[-4:3]
[11,12]
>>> l[:3]           # grab the first three elements
[10,11,12]
>>> l[-2:]         # grab the last two elements
[13,14]
```

# Tuples (1)

- Tuples are a sequence of objects just like lists.
- Unlike lists, tuples are immutable objects.
- A good rule of thumb is to use lists whenever you need a generic sequence.

# Tuples (2)

- Tuple = constant list; items cannot be modified

```
>>> s1=[1.2, 1.3, 1.4]      # list
>>> s2=(1.2, 1.3, 1.4)     # tuple
>>> s2=1.2, 1.3, 1.4      # may skip parenthesis
>>> s1[1]=0                # ok
>>> s2[1]=0                # illegal
```

Traceback (innermost last):

```
  File "<pyshell#17>", line 1, in ?
```

```
    s2[1]=0
```

**TypeError: object doesn't support item assignment**

```
>>> s2.sort()
```

**AttributeError: 'tuple' object has no attribute 'sort'**

- You cannot append to tuples, but you can add two tuples to form a new tuple

# Dictionaries

- Dictionary = array with a text as index
- Also called *hash* or *associative array* in other languages
- Can store 'anything' :

```
prm['damping'] = 0.2                # number
```

```
def x3(x):
```

```
    return x*x*x
```

```
prm['stiffness'] = x3                # function object
```

```
prm['model1'] = [1.2, 1.5, 0.1]     # list object
```

- The text index is called *key*

# Dictionary Operations

- Dictionary = array with text indices (keys, even user-defined objects can be indices!)
- Also called hash or associative array
- Common operations:

```
d['mass']           # extract item corresp. to key 'mass'  
d.keys()            # return copy of list of keys  
d.get('mass',1.0)   # return 1.0 if 'mass' is not a key  
d.has_key('mass')   # does d have a key 'mass'?  
d.values()          # return a list of all values in the dictionary  
d.items()           # return list of (key,value) tuples  
d.copy()            # create a copy of the dictionary  
d.clear()           # remove all key/value pair from the dictionary  
del d['mass']        # delete an item  
len(d)              # the number of items
```

# Initializing Dictionaries

- Multiple items:

```
d = { 'key1' : value1, 'key2' : value2 }
```

```
# or
```

```
d = dict(key1=value1, key2=value2)
```

- Item by item (indexing):

```
d['key1'] = anothervalue1
```

```
d['key2'] = anothervalue2
```

```
d['key3'] = value2
```

# Dictionary Example (1)

```
# create an empty dictionary using curly brackets
>>> record = {}
>>> record['first'] = 'Jmes'
>>> record['last'] = 'Maxwell'
>>> record['born'] = 1831
>>> print record
{'first': 'Jmes', 'born': 1831, 'last': 'Maxwell'}
# create another dictionary with initial entries
>>> new_record = {'first': 'James', 'middle': 'Clerk'}
# now update the first dictionary with values from the new one
>>> record.update(new_record)
>>> print record
{'first': 'James', 'middle': 'Clerk', 'last': 'Maxwell', 'born': 1831}
```

## Dictionary Example (2)

- Problem: store MPEG filenames corresponding to a parameter with values 1, 0.1, 0.001, 0.00001

```
movies[1]           = 'heatsim1.mpeg'  
movies[0.1]        = 'heatsim2.mpeg'  
movies[0.001]      = 'heatsim5.mpeg'  
movies[0.00001]   = 'heatsim8.mpeg'
```

- Store compiler data:

```
g77 = {  
  'name'           : 'g77',  
  'description'    : 'GNU f77 compiler, v2.95.4',  
  'compile_flags' : ' -pg',  
  'link_flags'    : ' -pg',  
  'libs'          : '-lf2c',  
  'opt'           : '-O3 -ffast-math -funroll-loops'  
}
```

# Sample Dictionary

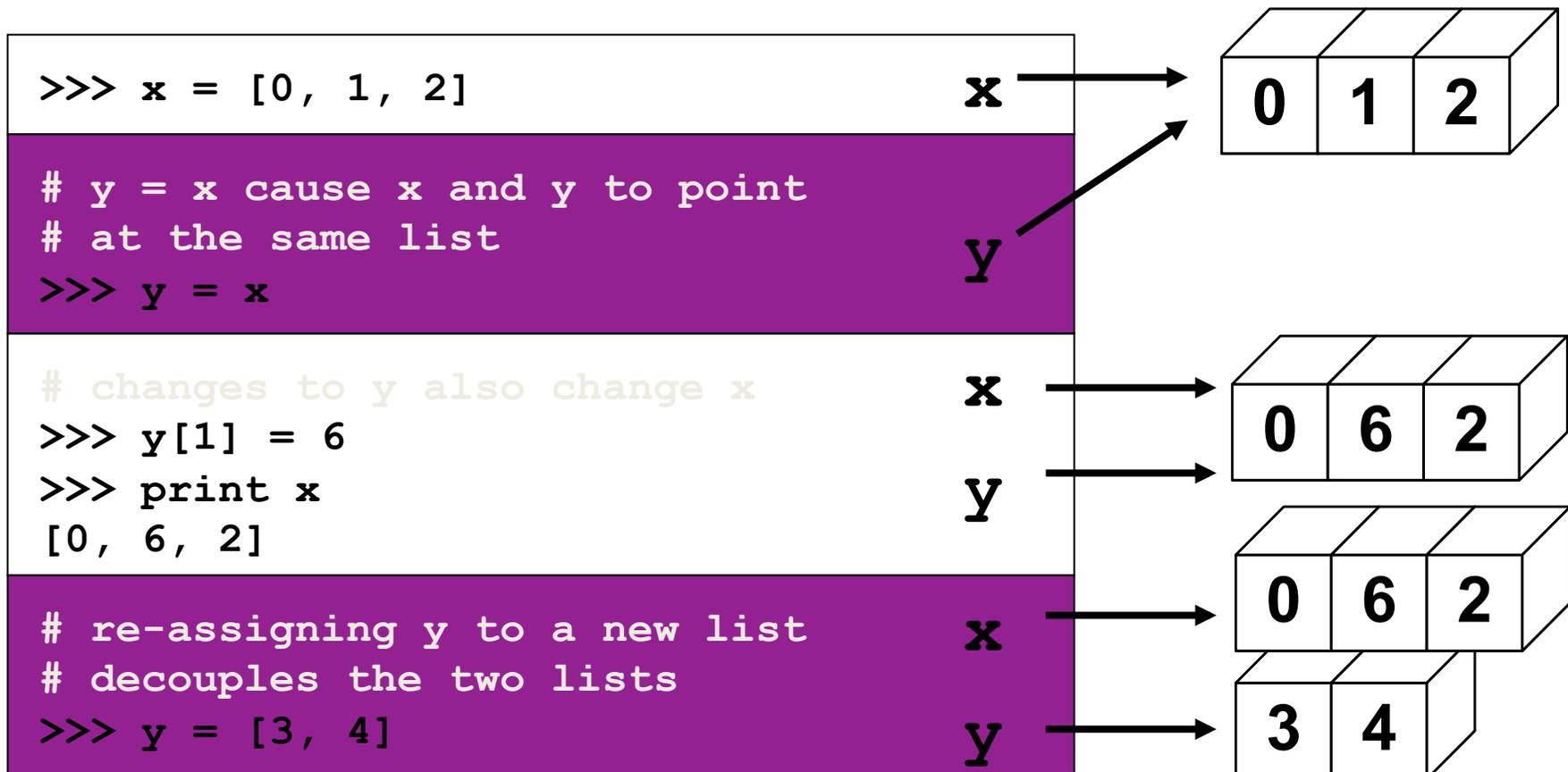
Check the file:

`sampleDictionary.py`

and run it

# Assignment

Assignment creates object references.



# Numpy

- **Numpy arrays**
- **Complex Number Computing**
- **In-Place Array Arithmetics**
- **Vectorization**
- **Matrix Objects**
- **Random Numbers**
- **Basic Linear Algebra**
- **Numerical Solution of the Laplace Equation**
- **Numpy for Matlab Users**

# What is Numpy?

- Efficient array computing in Python
- Creating arrays
- Indexing/slicing arrays
- Random numbers
- Linear algebra
- (The functionality is close to that of Matlab)

# Using NumPy

- The critical thing to know is that Python **for** loops are *slow* – one should try to use array-operations as much as possible
- NumPy provides mathematical functions that operate on an entire array.

# Making Arrays

```
>>> from numpy import *
>>> n = 4
>>> a = zeros(n)           # one-dim. array of length n
>>> print a               # str(a), float (C double) is default type
[ 0.  0.  0.  0.]
>>> a                     # repr(a)
array([ 0.,  0.,  0.,  0.])
>>> p = q = 2
>>> a = zeros((p,q,3))    # p*q*3 three-dim. Array
>>> print a
[[[ 0.  0.  0.]
   [ 0.  0.  0.]]

 [[ 0.  0.  0.]
   [ 0.  0.  0.]]]
>>> a.shape               # a's dimension
(2, 2, 3)
```

# Making int, float, complex arrays

```
>>> a = zeros(3)
>>> print a.dtype # a's data
Typefloat64
>>> a = zeros(3, int)
>>> print a
[0 0 0]
>>> print a.dtype
Int32
>>> a = zeros(3, float32) # single precision
>>> print a
[ 0.  0.  0.]
>>> print a.dtype
float32
>>> a = zeros(3, complex)
>>> a
array([ 0.+0.j,  0.+0.j,  0.+0.j])
>>> a.dtype
dtype('complex128')
>>> x = zeros(a.shape, a.dtype)
```

# Array with Sequence of Numbers

- `linspace(a, b, n)` generates `n` uniformly spaced coordinates, starting with `a` and ending with `b`

```
>>> x = linspace(-5, 5, 11)
>>> print x
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.]
```

- A special compact syntax is available through the syntax

```
>>> a = r_[-5:5:11j]           # same as linspace(-1, 1, 11)
>>> print a
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.]
```

- `arange` works like `range` (`xrange`)

```
>>> x = arange(-5, 5, 1, float)
>>> print x           # upper limit 5 is not included!!
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.]
```

# Array Construction from a Python List

- `array(list, [datatype])` generates an array from a list:

```
>>> pl = [0, 1.2, 4, -9.1, 5, 8]
>>> a = array(pl)
```

- The array elements are of the simplest possible type:

```
>>> z = array([1, 2, 3])
>>> print z                                # int elements possible
[1 2 3]
>>> z = array([1, 2, 3], float)
>>> print z
[ 1.  2.  3.]
```

- A two-dim. array from two one-dim. lists:

```
>>> x = [0, 0.5, 1]; y = [-6.1, -2, 1.2]    # Python lists
>>> a = array([x, y])                       # form array with x and y as rows
```

- From array to list: `alist = a.tolist()`

# From “Anything” to a Numpy Array

- Given an object a,

```
a = asarray(a)
```

converts a to a NumPy array (if possible/necessary)

- Arrays can be ordered as in C (default) or Fortran:

```
a = asarray(a, order='fortran')
```

```
isfortran(a)
```

```
# returns True if a's order is fortran
```

- Use asarray to, e.g., allow flexible arguments in functions:

```
def myfunc(some_sequence, ...):
```

```
    a = asarray(some_sequence)
```

```
    # work with a as array
```

```
myfunc([1,2,3], ...)
```

```
myfunc((-1,1), ...)
```

```
myfunc(zeros(10), ...)
```

## Changing Array Dimensions

```
>>> a = array([0, 1.2, 4, -9.1, 5, 8])
>>> a.shape = (2,3)      # turn a into a 2x3 matrix
>>> a.size
6
>>> a.shape = (a.size,)  # turn a into a vector of length 6 again
>>> a.shape
(6,)
>>> a = a.reshape(2,3)   # same effect as setting a.shape
>>> a.shape(2, 3)
```

# Array Initialization from a Python Function

```
>>> def myfunc(i, j):  
...     return (i+1)*(j+4-i)  
...  
>>> # make 3x6 array where a[i,j] = myfunc(i,j):  
>>> a = fromfunction(myfunc, (3,6))  
>>> a  
array([[ 4.,  5.,  6.,  7.,  8.,  9.],  
       [ 6.,  8., 10., 12., 14., 16.],  
       [ 6.,  9., 12., 15., 18., 21.]])
```

# Basic Array Indexing

```
a = linspace(-1, 1, 6)
a[2:4] = -1          # set a[2] and a[3] equal to -1
a[-1] = a[0]        # set last element equal to first one
a[:] = 0            # set all elements of a equal to 0
a.fill(0)           # set all elements of a equal to 0

a.shape = (2,3)     # turn a into a 2x3 matrix
print a[0,1]        # print element (0,1)
a[i,j] = 10         # assignment to element (i,j)
a[i][j] = 10        # equivalent syntax (slower)
print a[:,k]        # print column with index k
print a[1,:]        # print second row
a[:,:] = 0          # set all elements of a equal to 0
```

# More Advanced Array Indexing

```
>>> a = linspace(0, 29, 30)
>>> a.shape = (5,6)
>>> a
array([[ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 6.,  7.,  8.,  9., 10., 11.],
       [12., 13., 14., 15., 16., 17.],
       [18., 19., 20., 21., 22., 23.],
       [24., 25., 26., 27., 28., 29.]])
>>> a[1:3,-1:2] # a[i,j] for i=1,2 and j=0,2,4
array([[ 6.,  8., 10.],
       [12., 14., 16.]])
>>> a[:,2:-1:2] # a[i,j] for i=0,3 and j=2,4
array([[ 2.,  4.],
       [20., 22.]])
>>> i = slice(None, None, 3); j = slice(2, -1, 2)
>>> a[i,j]
array([[ 2.,  4.],
       [20., 22.]])
```

# Array Slicing

**SLICING WORKS MUCH LIKE  
STANDARD PYTHON SLICING**

```
>>> a[0,3:5]
array([3,4])
```

```
>>> a[4:,4:]
array([[44, 45],
       [54, 55] ])
```

```
>>> a[:,2]
array([2,12,22,32,42,52])
```

**STRIDES ARE ALSO POSSIBLE**

```
>>> a[2::2,::2]
array([[20, 22, 24],
       [40, 42, 44] ])
```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55



# Integer Arrays as Indices

- An integer array or list can be used as (vectorized) index

```
>>> a = linspace(1, 8, 8)
>>> aarray([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.])
>>> a[[1,6,7]] = 10
>>> a
array([  1., 10.,  3.,  4.,  5.,  6., 10., 10.])
>>> a[range(2,8,3)] = -2
>>> aarray([  1., 10., -2.,  4.,  5., -2., 10., 10.])
>>> a[a < 0]          # pick out the negative elements of a
array([-2., -2.])
>>> a[a < 0] = a.max()
>>> a
array([  1., 10., 10.,  4.,  5., 10., 10., 10.])
```

- Such array indices are important for efficient vectorized code

# Loop over Arrays (1)

- Standard loop over each element:

```
for i in xrange(a.shape[0]):
    for j in xrange(a.shape[1]):
        a[i,j] = (i+1)*(j+1)*(j+2)
        print 'a[%d,%d]=%g ' % (i,j,a[i,j]),
    print # newline after each row
```

- A standard for loop iterates over the first index:

```
>>> print a
[[ 2.  6. 12.]
 [ 4. 12. 24.]]
>>> for e in a:
...     print e
...
[ 2.  6. 12.]
[ 4. 12. 24.]
```

## Loop over Arrays (2)

- View array as one-dimensional and iterate over all elements:

```
for e in a.flat:  
    print e
```

- For loop over all index tuples and values:

```
>>> for index, value in ndenumerate(a):  
...     print index, value  
...  
(0, 0) 2.0  
(0, 1) 6.0  
(0, 2) 12.0  
(1, 0) 4.0  
(1, 1) 12.0  
(1, 2) 24.0
```

# Array Computations

- Arithmetic operations can be used with arrays:  
`b = 3*a - 1` # a is array, b becomes array

- Array operations are much faster than element-wise operations:

```
import time # module for measuring CPU time
a = linspace(0, 1, 1E+07) # create some array
t0 = time.clock()
b = 3*a - 1
t1 = time.clock() # t1-t0 is the CPU time of 3*a-1

for i in xrange(a.size):
    b[i] = 3*a[i] - 1
t2 = time.clock()
print '3*a-1: %g sec, loop: %g sec' % (t1-t0, t2-t1)
```

# In-Place Array Arithmetics

- Expressions like  $3*a-1$  generates temporary arrays
- With in-place modifications of arrays, we can avoid temporary arrays (to some extent)

```
b = a
```

```
b *= 3 # or multiply(b, 3, b)
```

```
b -= 1 # or subtract(b, 1, b)
```

Note: a is changed, use `b = a.copy()`

- In-place operations:

```
a *= 3.0 # multiply a's elements by 3
```

```
a -= 1.0 # subtract 1 from each element
```

```
a /= 3.0 # divide each element by 3
```

```
a += 1.0 # add 1 to each element
```

```
a **= 2.0 # square all elements
```

- Assign values to all elements of an existing array:

```
a[:] = 3*c - 1
```

# Math Functions & Array Arguments

```
# let b be an array
```

```
c = sin(b)
```

```
c = arcsin(c)
```

```
c = sinh(b)
```

```
# same functions for the cos and tan families
```

```
c = b**2.5 # power function
```

```
c = log(b)
```

```
c = exp(b)
```

```
c = sqrt(b)
```

# Other Useful Array Operations

# a is an array

a.clip(min=3, max=12)      # clip elements

a.mean(); mean(a)            # mean value

a.var(); var(a)              # variance

a.std(); std(a)              # standard deviation

median(a)

cov(x,y)                      # covariance

trapz(a)                      # Trapezoidal integration

diff(a)                      # finite differences (da/dx)

# more Matlab-like functions:

corrcoeff, cumprod, diag, eig, eye, fliplr, flipud, max, min, prod, ptp, rot90, squeeze, sum, svd,  
tri, tril, triu

# More Useful Array Methods and Attributes

```
>>> a = zeros(4) + 3
>>> a
array([ 3.,  3.,  3.,  3.])          # float data
>>> a.item(2)                        # more efficient than a[2]
3.0
>>> a.itemset(3,-4.5)                # more efficient than a[3]=-4.5
>>> a
array([ 3. ,  3. ,  3. , -4.5])
>>> a.shape = (2,2)
>>> a
array([[ 3. ,  3. ],
       [ 3. , -4.5]])
>>> a.ravel()                        # from multi-dim to one-dim
array([ 3. ,  3. ,  3. , -4.5])
>>> a.ndim                            # no of dimensions
2
>>> len(a.shape)                     # no of dimensions
2
>>> rank(a)                          # no of dimensions
2
>>> a.size                            # total no of elements
4
>>> b = a.astype(int)                # change data type
>>> b
array([3, 3, 3, 3])
```

# Vectorization (1)

- Loops over an array run slowly
- Vectorization = replace explicit loops by functions calls such that the whole loop is implemented in C (or Fortran)
- Explicit loops:  

```
r = zeros(x.shape, x.dtype)  
for i in xrange(x.size):  
    r[i] = sin(x[i])
```
- Vectorized version:  

```
r = sin(x)
```
- Arithmetic expressions work for both scalars and arrays
- Many fundamental functions work for scalars and arrays
- Ex: **x\*\*2 + abs(x)** works for **x** scalar or array

# Vectorization (2)

A mathematical function written for scalar arguments can (normally) take a array arguments:

```
>>> def f(x):
...     return x**2 + sinh(x)*exp(-x) + 1
...
>>> # scalar argument:
>>> x = 2
>>> f(x)
5.4908421805556333

>>> # array argument:
>>> y = array([2, -1, 0, 1.5])
>>> f(y)
array([ 5.49084218, -1.19452805,  1.          ,  3.72510647])
```

# Vectorization of Functions with if-tests

- Consider a function with an if test:

```
def somefunc(x):  
    if x < 0:  
        return 0  
    else:  
        return sin(x)
```

# or

```
def somefunc(x): return 0 if x < 0 else sin(x)
```

- This function works with a scalar  $x$  but not an array
- Problem:  $x < 0$  results in a boolean array, not a boolean value that can be used in the if test

```
>>> x = linspace(-1, 1, 3); print x[-1. 0. 1.]
```

```
>>> y = x < 0
```

```
>>> y
```

```
array([ True, False, False], dtype=bool)
```

```
>>> 'ok' if y else 'not ok' # test of y in scalar boolean context
```

```
...
```

```
ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

# Vectorization of Functions with if-tests

- Simplest remedy: call NumPy's `vectorize` function to allow array arguments to a function:

```
>>> somefuncv = vectorize(somefunc, otypes='d')
>>> # test:
>>> x = linspace(-1, 1, 3); print x[-1. 0. 1.]
>>> somefuncv(x)
array([ 0.          , 0.          , 0.84147098])
```

Note: The data type must be specified as a character

- The speed of `somefuncv` is unfortunately quite slow
- A better solution, using `where`:

```
def somefunc_NumPy2(x):
    x1 = zeros(x.size, float)
    x2 = sin(x)
    return where(x < 0, x1, x2)
```

# General Vectorization with if-else Tests

```
def f(x):                                # scalar x
    if condition:
        x = <expression1>
    else:
        x = <expression2>
    return x

def f_vectorized(x):                     # scalar or array x
    x1 = <expression1>
    x2 = <expression2>
    return where(condition, x1, x2)
```

# Vectorization via Slicing

- Consider a recursion scheme like  $u_{i+1} = \beta u_{i-1} + (1-2\beta)u_i + \beta u_{i+1}$  (which arises from a one-dimensional diffusion equation)
- Straightforward (slow) Python implementation:

```
n = size(u)-1
for i in xrange(1,n,1):
    u_new[i] = beta*u[i-1] + (1-2*beta)*u[i] + beta*u[i+1]
```

- Slices enable us to vectorize the expression:

```
u[1:n] = beta*u[0:n-1] + (1-2*beta)*u[1:n] + beta*u[2:n+1]
```

# Matrix Objects (1)

- NumPy has an array type, matrix, much like Matlab's array type

```
>>> x1 = array([1, 2, 3], float)
>>> x2 = matrix(x)                # or just mat(x)
>>> x2                            # row vector
matrix([[ 1.,  2.,  3.]])
>>> x3 = mat(x).transpose()      # column vector
>>> x3
matrix([[ 1.],
        [ 2.],
        [ 3.]])
>>> type(x3)
<class 'numpy.core.defmatrix.matrix' >
>>> isinstance(x3, matrix)
True
```

- Only 1- and 2-dimensional arrays can be matrix

## Matrix Objects (2)

- For matrix objects, the `*` operator means matrix-matrix or matrix-vector multiplication (not elementwise multiplication)

```
>>> A = eye(3)                # identity matrix
>>> A = mat(A)                # turn array to matrix
>>> A
matrix([[ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])
>>> y2 = x2*A                 # vector-matrix product
>>> y2
matrix([[ 1.,  2.,  3.]])
>>> y3 = A*x3                 # matrix-vector product
>>> y3
matrix([[ 1.],
        [ 2.],
        [ 3.]])
```

# Example 1

```
a = array([[1,2],[3,4]])
```

```
a
```

```
m = mat(a)
```

```
m
```

```
a[0]
```

```
m[0]
```

```
a*a
```

```
m*m
```

## Example 2

```
x = array([1,0,2,-1,0,0,8])
```

```
indices = x.nonzero()
```

Indices

```
x[indices]
```

```
indices = (x > -1).nonzero()
```

```
x[indices]
```

## Example 3

```
a = array([1,2,3])
```

```
a.prod()
```

```
prod(a)
```

```
b = array([[1,2,3],[4,5,6]])
```

```
b.prod(dtype=float)
```

```
b.prod(axis=0)
```

```
b.prod(axis=1)
```

# NumPy for Matlab Users

- ✓ In NumPy, operations are elementwise by default
- ✓ There is a matrix type for linear algebra (subclass of array)
- ✓ Indexing starts at 0 in NumPy
- ✓ Using Python with NumPy gives more programming power
- ✓ Function definition in Matlab has many restrictions
- ✓ NumPy/SciPy is free but still widely used
- ✓ Matlab has lots of 'toolboxes' for specific tasks (much less in NumPy/SciPy)
- ✓ There are many packages for plotting in Python that are as good as Matlab

# Matlab/Numpy Equivalence (1)

## Matlab

```
a = [1 2 3; 4 5 6]
a(end)
a(2,5)
a(2,:)
a(1:5,:)
a(end-4:end,:)
a(1:3,5:9)
a(1:2:end,:)
a(end:-1:1,:) or flipud(a)
a.'
a'
a * b
a .* b
a./b
```

## NumPy

```
a = array([[1.,2.,3.],[4.,5.,6.]])
a[-1]
a[1,4]
a[1] or a[1,:]
a[0:5] or a[:5] or a[0:5,:]
a[-5:]
a[0:3][:,4:9]
a[:,2:]
a[:,:-1,:]
a.transpose() or a.T
a.conj().transpose() or a.conj().T
dot(a,b)
a * b
a/b
```

# Matlab/Numpy Equivalence (3)

## Matlab

diag(a)  
diag(a,0)  
rand(3,4)  
linspace(1,3,4)  
[x,y]=meshgrid(0:8,0:5)  
repmat(a, m, n)  
[a b]  
[a; b]  
max(max(a))  
max(a)  
max(a,[],2)  
max(a,b)  
norm(v)

## NumPy

diag(a) or a.diagonal()  
diag(a,0) or a.diagonal(0)  
random.rand(3,4)  
linspace(1,3,4)  
mgrid[0:9.,0:6.]  
tile(a, (m, n))  
concatenate((a,b),1) or hstack((a,b)) or c [a,b[a; b]  
concatenate((a,b)) or vstack((a,b)) or r [a,b]  
a.max()  
a.max(0)  
a.max(1)  
where(a>b, a, b)  
sqrt(dot(v,v)) or linalg.norm(v)

# Matlab/Numpy Equivalence (4)

## Matlab

inv(a)

pinv(a)

a\b

b/a

[U,S,V]=svd(a)

chol(a)

[V,D]=eig(a)

[Q,R,P]=qr(a,0)

[L,U,P]=lu(a)

conjgrad

fft(a)

ifft(a)

sort(a)

sortrows(a,i)

## NumPy

linalg.inv(a)

linalg.pinv(a)

linalg.solve(a,b)

Solve  $a.T x.T = b.T$  instead

(U, S, V) = linalg.svd(a)

linalg.cholesky(a)

linalg.eig(a)

(Q,R)=Sci.linalg.qr(a)

(L,U)=linalg.lu(a) or (LU,P)=linalg.lu factor(a)

Sci.linalg.cg

fft(a)

ifft(a)

sort(a) or a.sort()

a[argsort(a[:,0],i)]

# Matlab/Numpy Equivalence (3)

## Matlab

```
a.^3  
find(a>0.5)  
a(a<0.5)=0  
a(:) = 3  
y=x  
y=x(2,:)   
y=x(:)  
1:10  
0:9  
zeros(3,4)  
zeros(3,4,5)  
ones(3,4)  
eye(3)
```

## NumPy

```
a**3  
where(a>0.5)  
a[a<0.5]=0  
a[:] = 3  
y = x.copy()  
y = x[2,:].copy()  
y = x.flatten(1)  
arange(1.,11.) or r [1.:11.]  
arange(10.) or r [:10.]  
zeros((3,4))  
zeros((3,4,5))  
ones((3,4))  
eye(3)
```

# Sample Matrix Multiplication

Given two nxn matrices A and B, we want to compute:

$$C = AxB$$

A and B have randomly generated entries.

Check the files:

```
matMultiPython.py    # Multiply two matrices using do loops  
matMultiNumpy.py    # Multiply two matrices using Numpy functions
```

# Results

Timing numbers:

<https://modelingguru.nasa.gov/clearspace/docs/DOC-1762>

	n=1000	n=1200	n=1500
Python			
NumPy	6.1	10.48	29.31
Matlab	0.5480	0.714639	0.9695
Fortran	1.692105	3.228201	8.960560
Blas	1.692105	3.284205	9.384586
C			
C++	1.75591	3.41267	9.32094

# Random Numbers

- Drawing scalar random numbers:

```
import random
random.seed(2198) # control the seed
print 'uniform random number on (0,1):', random.random()
print 'uniform random number on (-1,1):', random.uniform(-1,1)
print 'Normal(0,1) random number:', random.gauss(0,1)
```

- Vectorized drawing of random numbers (arrays):

```
from numpy import random
random.seed(12) # set seed
u = random.random(n) # n uniform numbers on (0,1)
u = random.uniform(-1, 1, n) # n uniform numbers on (-1,1)
u = random.normal(m, s, n) # n numbers from N(m,s)
```

- Note that both modules have the name random! A remedy:

```
import random as random_number # rename random for scalars
from numpy import * # random is now numpy.random
```

# Basic Linear Algebra

NumPy contains the *linalg* module for:

- solving linear systems
- computing the determinant of a matrix
- computing the inverse of a matrix
- computing eigenvalues and eigenvectors of a matrix
- solving least-squares problems
- computing the singular value decomposition of a matrix
- computing the Cholesky decomposition of a matrix

# numpy.linalg

```
cholesky(A)                # Cholesky decomposition
qr(a[, mode])              # Compute the qr factorization of a matrix
svd(a[, full_matrices, compute_uv]) # Singular Value Decomposition
eig(A)                     # Compute the eigenvalues and right eigenvectors of
                            # a square array
eigvals(A)                 # Compute the eigenvalues of a general matrix.
det(A)                     # Compute the determinant of a matrix
matrix_power(M, n)         # Raise a square matrix to the (integer) power n
solve(A, b)                # Solve a linear matrix equation, or system of linear
                            # scalar equations.
inv(A)                     # Compute the (multiplicative) inverse of a matrix
```

# A Linear Algebra Session

```
b = dot(A, x)                # matrix vector product
y = linalg.solve(A, b)      # solve A*y = b
if allclose(x, y, atol=1.0E-12, rtol=1.0E-12):
    print 'correct solution!'

d = linalg.det(A)
B = linalg.inv(A)

# check result:
R = dot(A, B) - eye(n)      # residual
R_norm = linalg.norm(R)     # frobenius norm of matrix R
print 'Residual R = A*A-inverse - I:', R_norm

A_eigenvalues = linalg.eigvals(A) # eigenvalues only
A_eigenvalues, A_eigenvectors = linalg.eig(A)

for e, v in zip(A_eigenvalues, A_eigenvectors):
    print 'eigenvalue %g has corresponding vector\n%s' % (e, v)

# file linAlgSession.py
```

# Jacobi Iterations

We want to find the numerical solution of the 2D Laplace equation:

$$u_{xx} + u_{yy} = 0$$

We use the Jacobi iterative solver.

# Schemes

We use two schemes:

$$u_{i,j} = (u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1}) / 4$$

$$u_{i,j} = (4(u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1}) + u_{i-1,j-1} + u_{i+1,j-1} + u_{i+1,j-1} + u_{i+1,j+1}) / 20$$

# Results-Scheme 1

Timing numbers:

	n=50	n=100
Python	31.9193	509.859
Numpy	0.41324	4.19535
Matlab	0.460252	5.483992
Fortran	0.036	0.524

## Results – Scheme 2

Timing numbers:

	n=50	n=100
Python	46.15203	751.783
Numpy	0.610216	6.38891
Matlab	0.640044	6.531990
Fortran	0.080	1.176

# SciPy

- `scipy.interpolate`
- `scipy.optimize`
- `scipy.integrate`
- `scipy.stats`
- `scipy.signal`
- `scipy.fft`

# Useful References

- **How to Think Like a Computer Scientist: Learning with Python 2nd Edition**, Jeffrey Elkner, Allen B. Downey, and Chris Meyers  
<http://openbookproject.net//thinkCSpy/>
- **Dive Into Python: Python from Novice to Pro**, Mark Pilgrim  
<http://diveintopython.org/>

# What is SciPy?

- Collection of mathematical algorithms and convenience functions built on the Numeric extension for Python
- Adds significant power to the interactive Python session
- Can become a data-processing and system-prototyping environment

# What SciPy Can Do

- **stats** – Statistical Functions
- **signal** – Signal Processing Tools
- **linalg** – Linear Algebra Tools
- **linsolve** – Linear Solvers
- **sparse** – Sparse Matrix
- **fftpack** – Discrete Fourier Transform Algorithms
- **ndimage** – n-dimensional Image Package
- **io** – Data Input and Output
- **integrate** – Integration Routines
- **interpolate** – Interpolation Tools

## SciPy and Numpy

- The SciPy library is built to work with NumPy arrays
- Depends on NumPy for array manipulations

# Loading SciPy

- Loading the SciPy module:

```
import scipy
```

- The following command will import all SciPy functions:

```
from scipy import *
```

- Help on SciPy

```
scipy.info(scipy)
```

```
help(scipy)
```

# scipy.interpolate

Two general interpolation facilities:

1. One class that performs 1D linear interpolation (**interp1d**)
2. Another (based on FITPACK) which provides 1D and 2D cubic-spline interpolations (**splrep**, **splev**, **bisplrep**, **bisplev**)

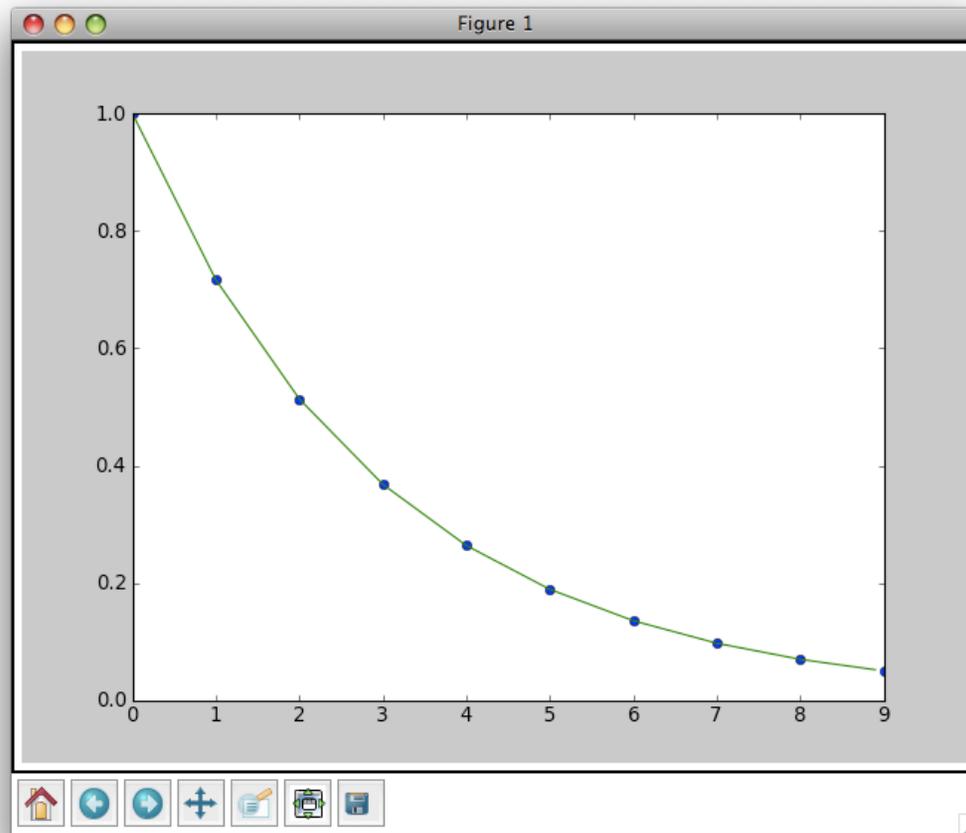
# scipy.interpolate Syntax

```
f = interp1d(x,y)           # 1D linear interpolation
```

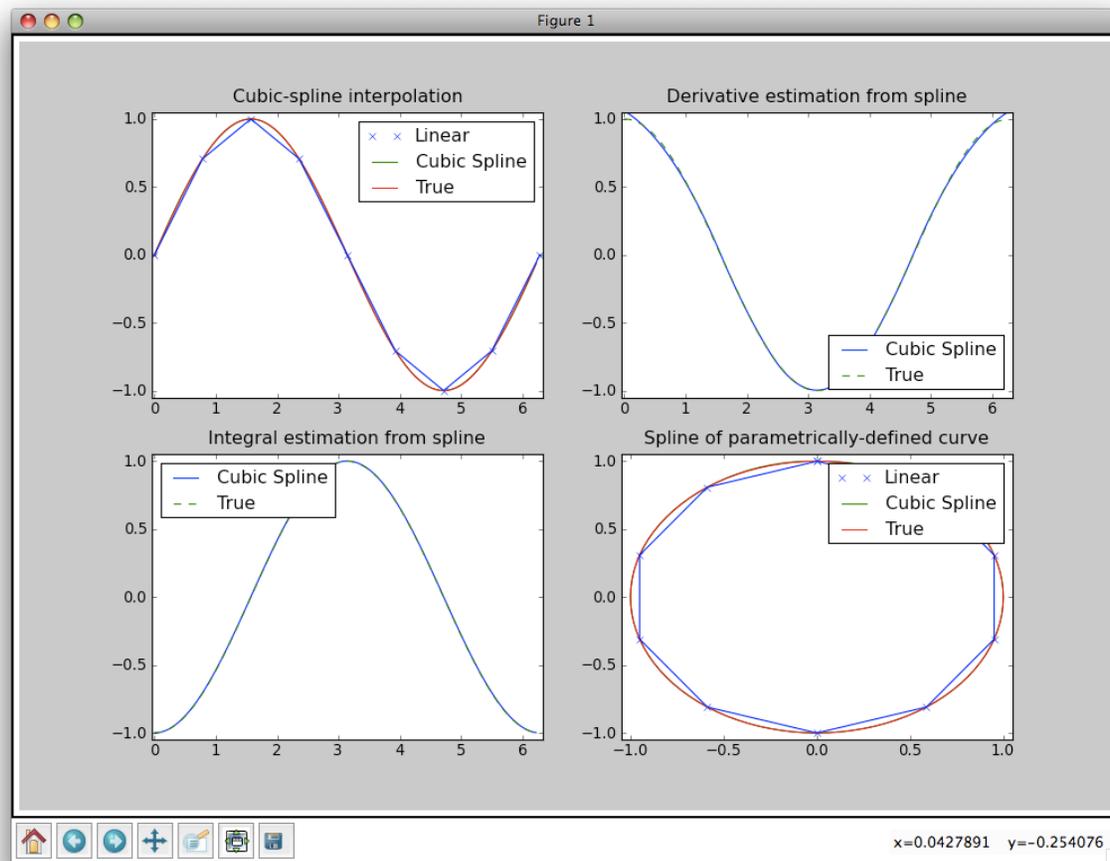
```
tck = splrep(x,y, k=n)     # B-spline representation of 1-D curve  
ynew = splev(xnew,tck,der=n) # evaluate the value of the smoothing  
                               # polynomial and it's derivatives
```

```
tck = bisplrep(x,y,z)     # compute a B-spline representation  
                           # of the surface  
znew = bisplev(xnew,ynew,tck) # spline function values
```

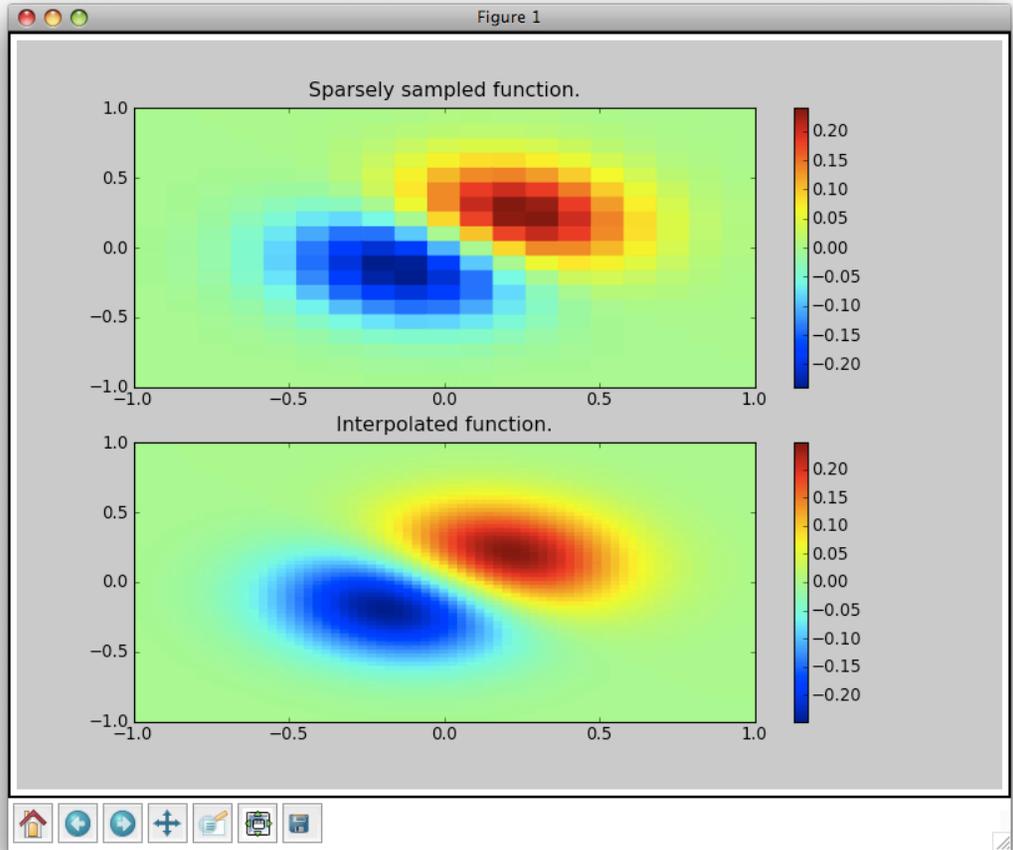
# 1D Linear Interpolation



# 1D Cubic Spline Interpolation



# 2D Cubic Spline Interpolation



# scipy.optimize

A collection of general-purpose optimization routines. We can mention:

<code>fminbound</code>	Bounded minimization for scalar functions
<code>fsolve</code>	Find the roots of a function
<code>fmin</code>	Minimize a function using the downhill simplex algorithm
<code>fixed_point</code>	Find the point where $\text{func}(x) == x$
<code>leastsq</code>	Minimize the sum of squares of a set of equations

# Bessell Functions and their Max

We want to plot a set of Bessell functions together with their maximum values.

```
x = arange(0,10,0.01)
```

```
for k in arange(0.5,5.5):
```

```
    y = special.jv(k,x)
```

```
    plt.plot(x,y)
```

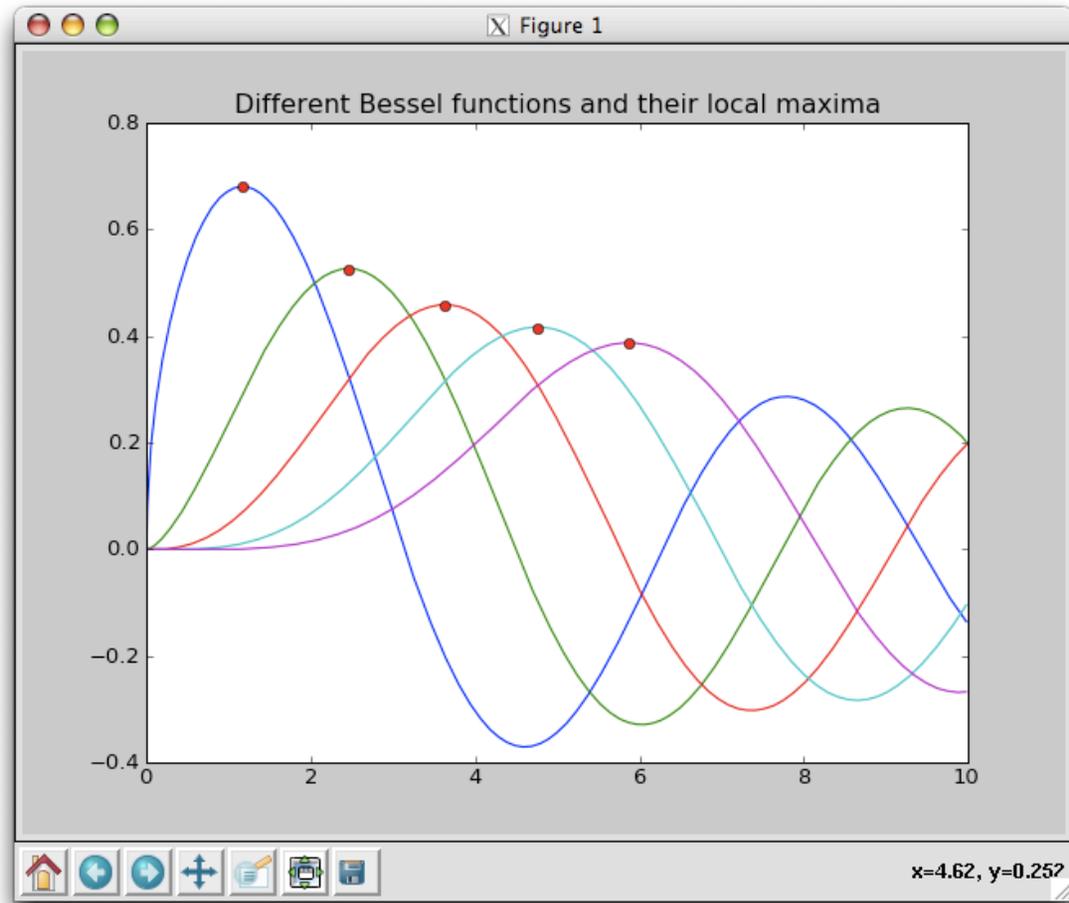
```
    f = lambda x: -special.jv(k,x)
```

```
    x_max = optimize.fminbound(f,0,6)
```

```
    plt.plot([x_max], [special.jv(k,x_max)],'ro')
```

```
    # File besselfunctions.py
```

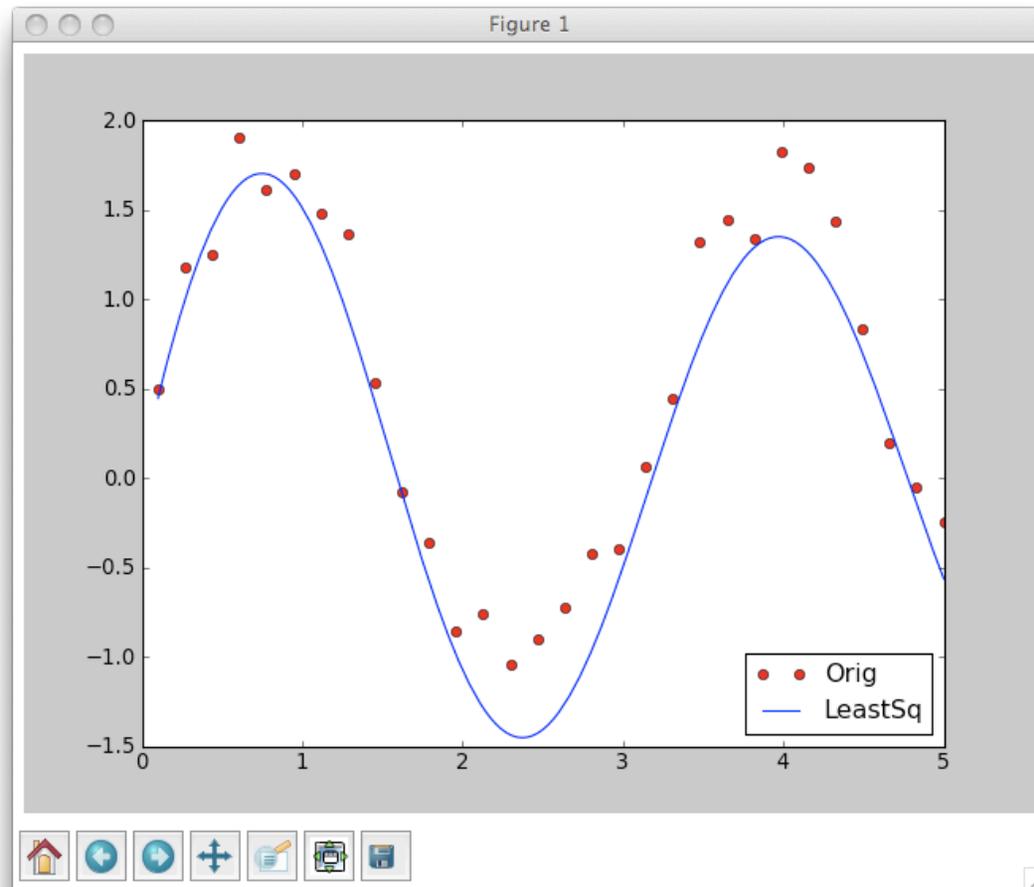
# Plots of Bessel Functions



## leastsq for Least Square Approx.

```
leastsq(efunc, x0, args=(x,y))
```

# Plot for Least Square



## fsolve for Root Finding

Assume that we want to solve the equations:

$$x + 2 \cos(x) = 0$$

$$\begin{cases} x_0 \cos(x_1) = 4 \\ x_0 x_1 - x_1 = 5 \end{cases}$$

# Code with fsolve

```
from scipy.optimize import fsolve
```

```
def func(x):
```

```
    return x + 2*scipy.cos(x)
```

```
def func2(x):
```

```
    out = [x[0]*scipy.cos(x[1]) - 4]
```

```
    out.append(x[1]*x[0] - x[1] - 5)
```

```
    return out
```

```
x0 = fsolve(func, 0.3)
```

```
print x0
```

```
x02 = fsolve(func2, [1, 1])
```

```
print x02
```

```
# file rootfindings.py
```

# Minimization Problem

Assume that we want to minimize the function:

$$f(x) = \sum_{i=1}^{N-1} 100(x_i - x_{i-1}^2)^2 + (1 - x_{i-1})^2$$

## Code with fmin

```
from scipy.optimize import fmin
def rosen(x):
    """The Rosenbrock function"""
    return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x[:-1])**2.0)

x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
xopt = fmin(rosen, x0, xtol=1e-8)

# file rosenbrockfunction.py
```

## Code with fixed\_point

```
from scipy.optimize import fixed_point

def func(x, c1, c2):
    return sqrt(c1/(x+c2))

c1 = array([10,12.])
c2 = array([3, 5.])
print fixed_point(func, [1.2, 1.3], args=(c1,c2))
```

# scipy.integrate

```
quad          -- General purpose integration.
dblquad       -- General purpose double integration.
tplquad       -- General purpose triple integration.
fixed_quad    -- Integrate func(x) using Gaussian quadrature of order n.
quadrature    -- Integrate with given tolerance using Gaussian quadrature.
romberg       -- Integrate func using Romberg integration.

trapz         -- Use trapezoidal rule to compute integral from samples.
cumtrapz     -- Use trapezoidal rule to cumulatively compute integral.
simps        -- Use Simpson's rule to compute integral from samples.
romb         -- Use Romberg Integration to compute integral from
              (2**k + 1) evenly-spaced samples.
odeint       -- General integration of ordinary differential equations.
ode          -- Integrate ODE using VODE and ZVODE routines.
```

# ODE

Assume that we want to solve the equation:

$$x''(t) + \mu x'(t)(x(t)^2 - 1) + x(t) = 0$$

It can be transformed into:

$$\begin{cases} x' = y \\ y' = -x + \mu y(1 - x^2) \end{cases}$$

# Code for ODE

```
import matplotlib.pyplot as plt
import scipy
from scipy import integrate

def f_1(y,t):
    return[y[1],-y[0]-10*y[1]*(y[0]**2-1)]

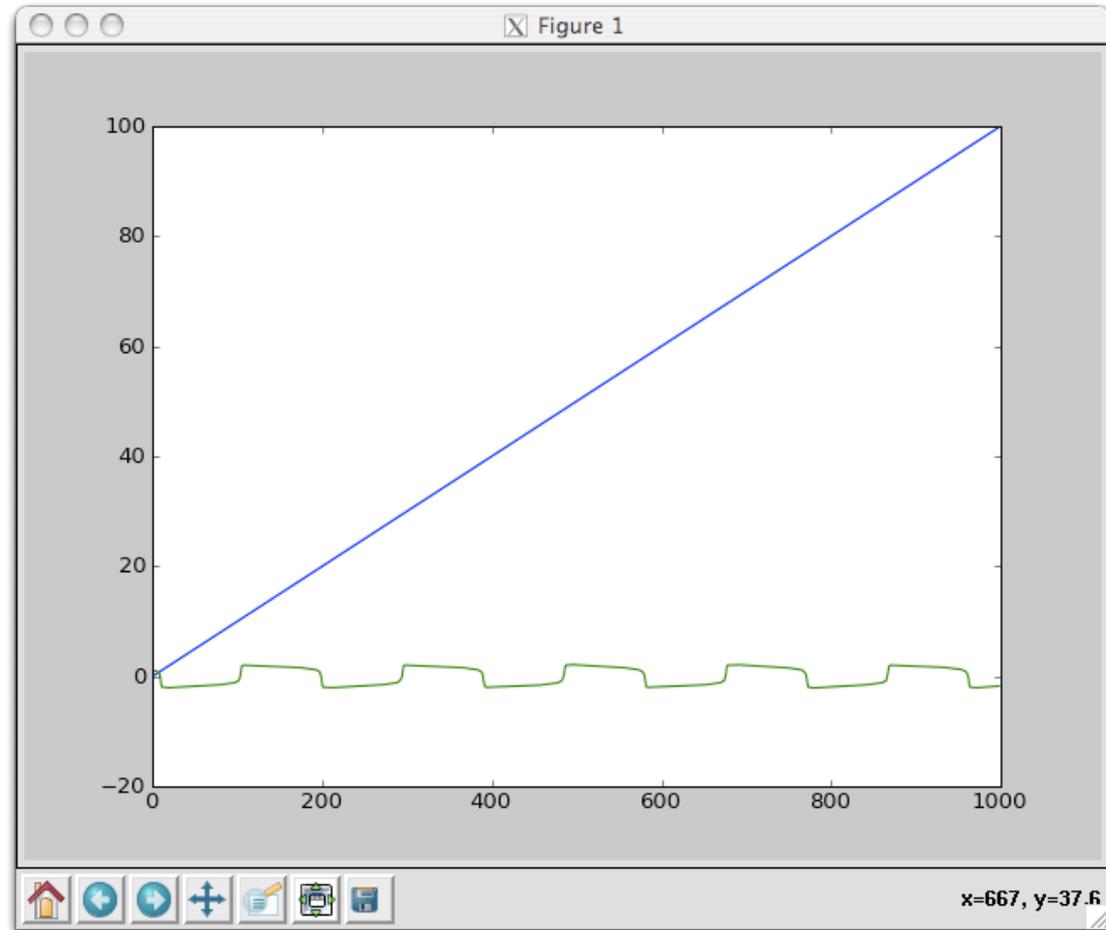
def j_1(y,t):
    return [ [0, 1.0],[-2.0*10*y[0]*y[1]-1.0,-10*(y[0]*y[0]-1.0)] ]

x= scipy.arange(0,100,.1)

y=integrate.odeint(f_1,[1,0],x,Dfun=j_1)

p=[((x[i],y[i][0])) for i in range(len(x))]
plt.plot(p)
plt.show()
```

# Picture



# scipy.stats

The package **scipy.stats** *provides tools for statistical analysis:*

- *More than 84 continuous distributions.*
- More than 12 discrete distributions
- Tools for manipulating them:
  - Statistical functions
  - Statistical tests
  - Statistical models

# Continuous Probability Distributions

norm	alpha	anglit	arcsine	beta	
betaprime	bradford	burr	fisk	cauchy	
chi	chi2	cosine	dgamma	dweibull	
erlang	expon	exponweib	exponpow	fatiguelife	
foldcauchy	f	foldnorm	frechet_r	frechet_l	
genlogistic	genpareto	genexpon	genextreme	gausshyper	
gamma	gengamma	genhalflogistic	gompertz	gumbel_r	
gumbel_l	halfcauchy	halflogistic	halfnorm	hypsecant	
invgamma	invnorm	invweibull	johnsonsb	johnsonsu	
laplace	logistic	loggamma	loglaplace	lognorm	
gilbrat	lomax	maxwell	mielke	nakagami	
ncx2	ncf	t	nct	pareto	
powerlaw	powerlognorm	powernorm	rdist	reciprocal	
rayleigh	rice	recipinvgauss	semicircular	triang	
truncexpon	truncnorm	tukeylambda	uniform	von_mises	
wald	weibull_min	weibull_max	wrapcauchy	ksone	
kstwobign					

# Syntax

**generic.pdf(x,<shape(s)>,loc=0,scale=1)**

probability density function

**generic.cdf(x,<shape(s)>,loc=0,scale=1)**

cumulative density function

**generic.ppf(q,<shape(s)>,loc=0,scale=1)**

percent point function (inverse of cdf --- percentiles)

**generic.rvs(<shape(s)>,loc=0,scale=1,size=1)**

random variates

**generic.stats(<shape(s)>,loc=0,scale=1,moments='mv')**

mean('m'), variance('v'), skew('s'), and/or kurtosis('k')

## Location and Scale

- Every distribution has location and scale parameters.
- $\text{pdf}(x, \text{loc}, \text{scale}) \rightarrow \text{pdf}((x-\text{loc})/\text{scale})/\text{scale}$

# Examples

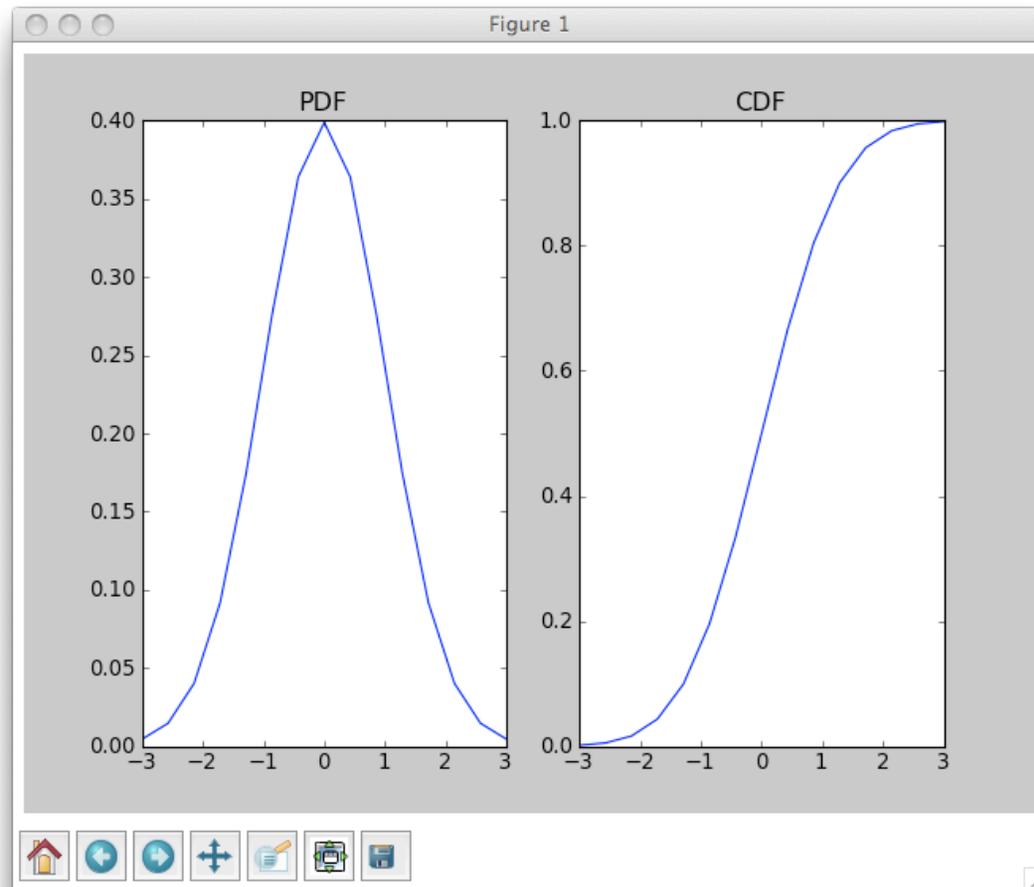
```
from scipy import stats

x = np.linspace(-3.0, 3.0, 15)
q = np.linspace(0.0, 1.0, 15)

stats.norm.pdf(x, loc=0.0, scale=1.0)
stats.norm.cdf(x, loc=0.0, scale=1.0)
stats.norm.ppf(q, loc=0.0, scale=1.0)
stats.norm.stats(loc=0.0, scale=1.0)
stats.norm.rvs(loc=0.0, scale=1.0, size=15)

# statDistribution.py
```

# Plots of Distributions



# Discrete Probability Distributions

binom	bernoulli	nbinom
geom	hypergeom	logser
poisson	planck	boltzmann
randint	zipf	dlaplace

Here we use  $\text{pmf}(x)$  instead of  $\text{pdf}(x)$ .  
probability mass function

# Summary Statistics

```
x = stats.norm.rvs(size=1000)
```

```
x.mean(); np.mean(x)
```

```
x.std(); np.std(x)
```

```
x.var(); np.var(x)
```

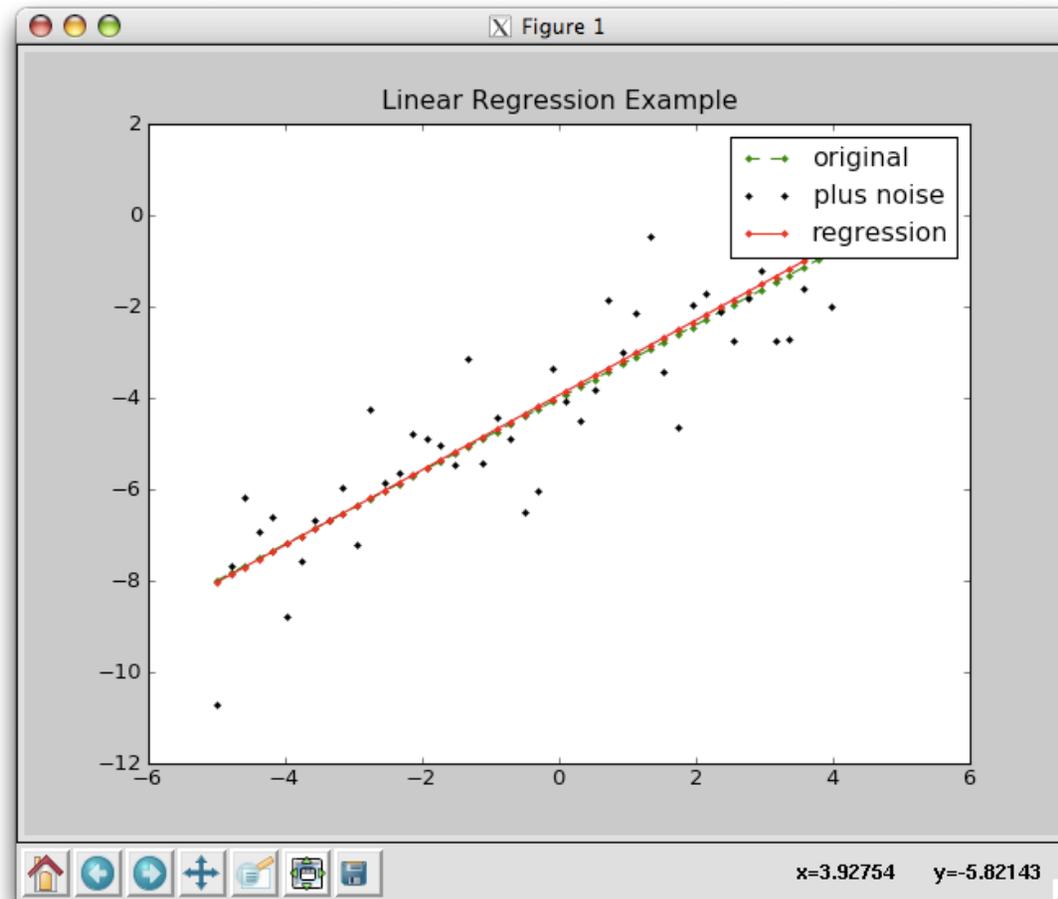
```
np.median(x)
```

```
stats.mode(stats.geom.rvs(0.1, size=1000))
```

## Examples with `scipy.stats`

- Linear Regression: `linearRegression.py`
- Example of distribution: `distributionExample.py`
- Computation of mean, std: `statEstimatorsSample.py`

# Graph for Linear Regression



# scipy.fftpack

## Discrete Fourier Transform Algorithms

- `fft`, `ifft`, `fft2`, `ifft2`, `fftn`, `ifftn`
- `fftshift`, `ifftshift`, `fftfreq`

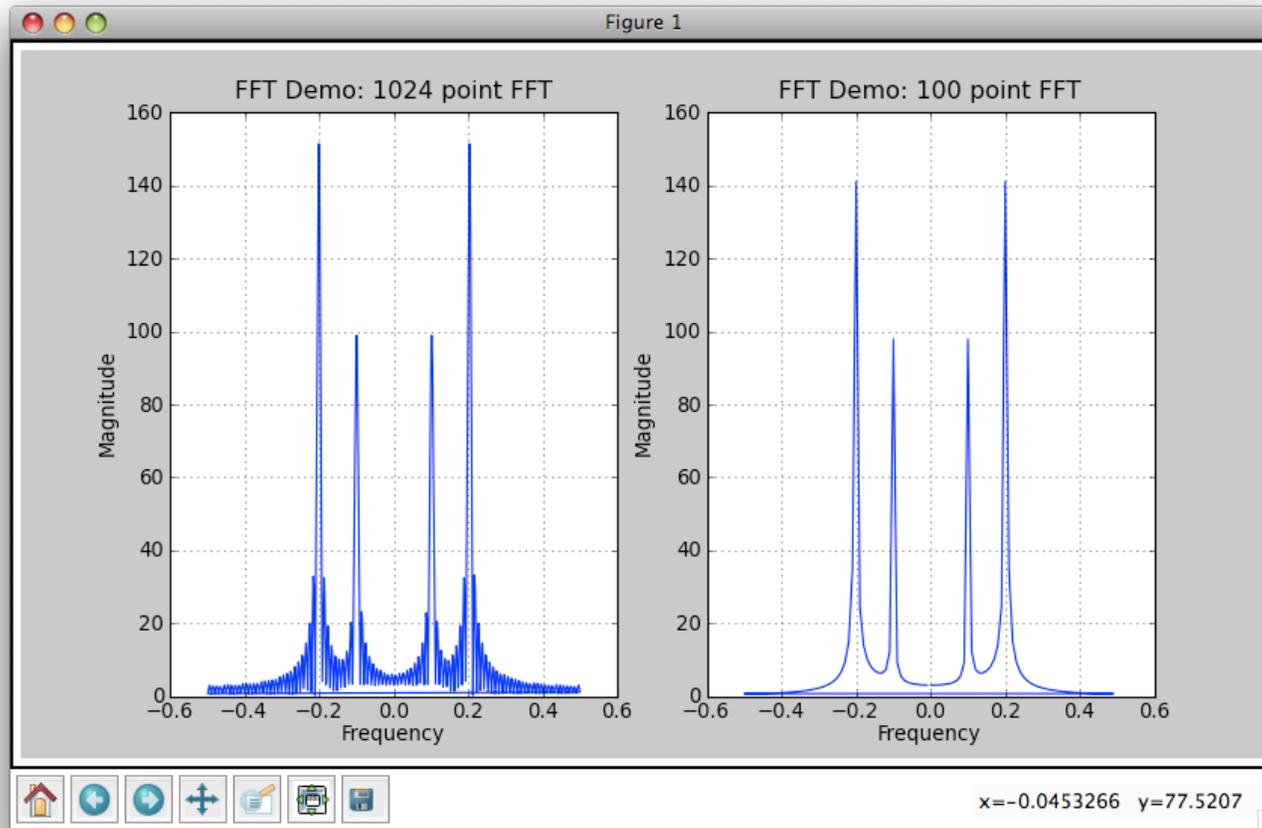
## Sample FFT Session

```
import matplotlib.pyplot as plt
from scipy import *
from scipy import *
from scipy.fftpack import fftshift, fftfreq

x = r_[0:1:100j]
y = 2*sin(2*pi*10*x) + 3*cos(2*pi*20*x)

Y02 = fft(y, 1024)
w = fftfreq(1024)
plt.plot(w,abs(Y02))
```

# FFT Demo



# scipy.signal

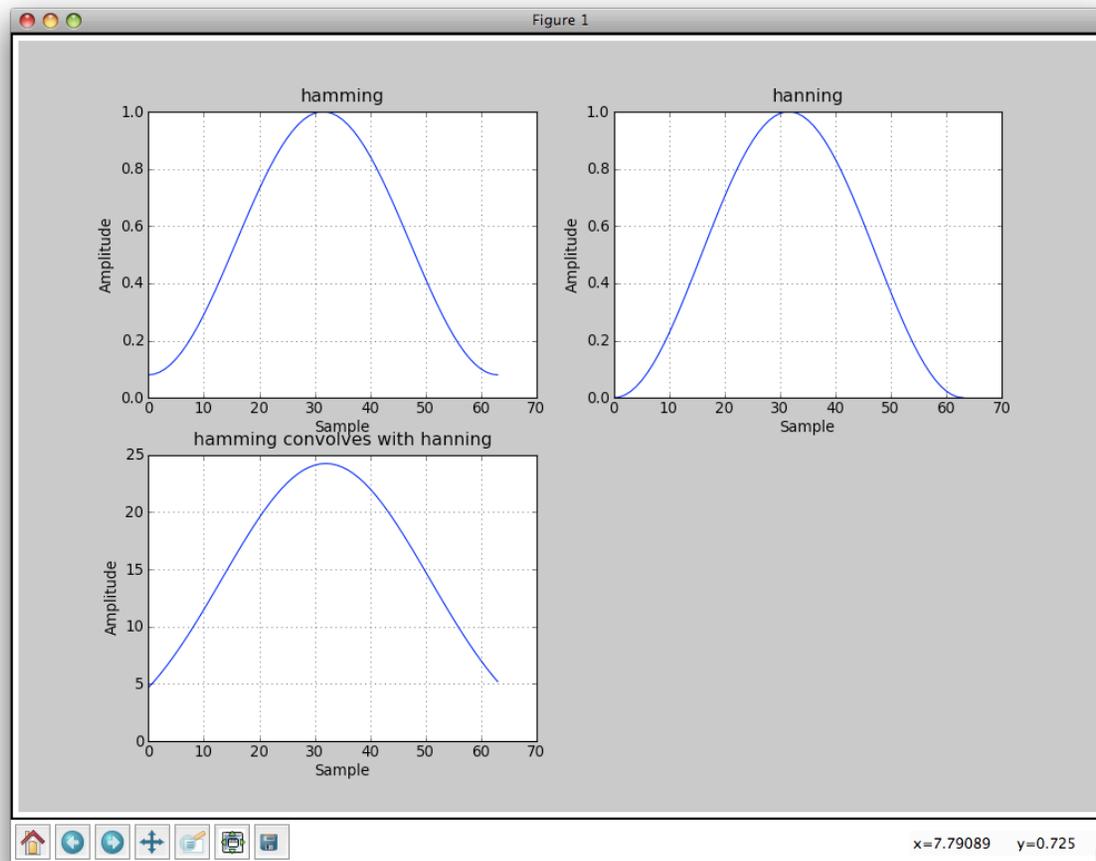
Provides functions for:

- Convolution
- B-Splines
- Filtering & Filter Design
- Linear Systems
- Window Functions
- Wavelets

## scipy.signal convolve

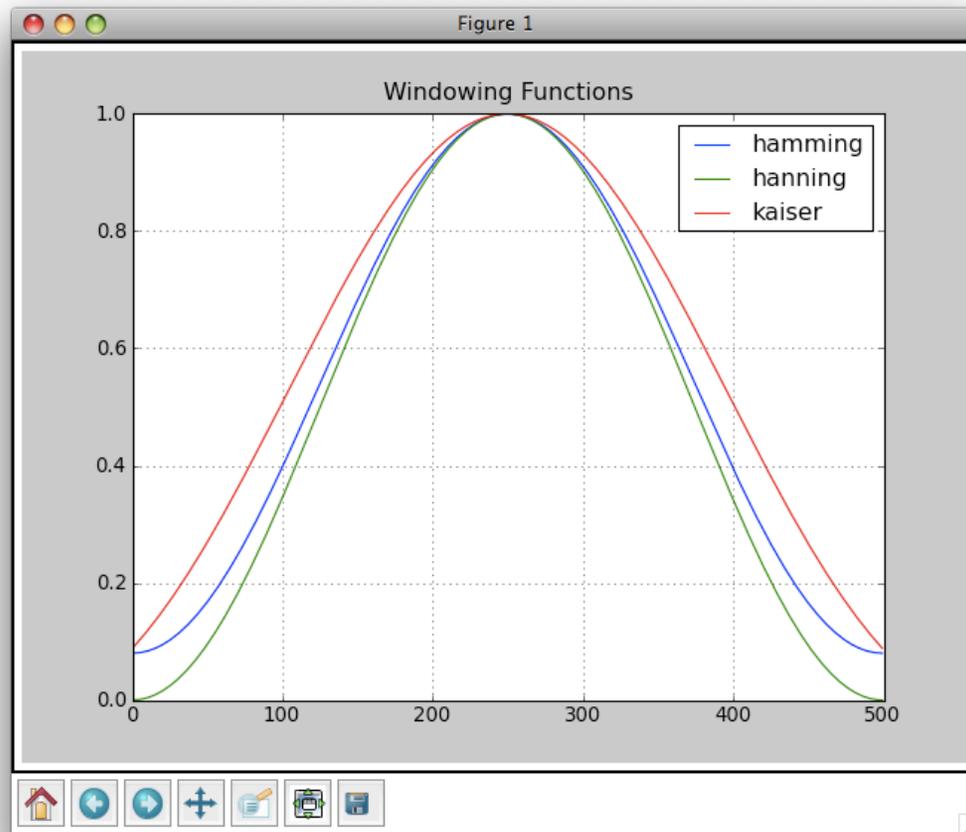
```
from scipy import *  
from scipy import signal  
  
n = 64  
x = linspace(0,n-1,n)  
  
y01 = hamming(n)  
y02 = hanning(n)  
  
z01 = signal.convolve(y01, y02, mode='same')
```

# Sample Convolution





# Sample Windows



# Matplotlib

- **Components of Matplotlib**
- **Plotting 1D & 2D Data**
- **Basemap Toolkit**

# Pointers

Video Presentation

[http://videolectures.net/mloss08\\_hunter\\_mat/](http://videolectures.net/mloss08_hunter_mat/)

User's Guide

<http://mural.uv.es/parmur/matplotlib.pdf>

Image Gallery

<http://matplotlib.sourceforge.net/gallery.html>

# What is Matplotlib?

- Library for making 2D plots of arrays in Python
- Makes heavy use of Numpy and other extension code to provide good performance
- Can be used to create plots with few commands

# What Can We Do?

You can generate plots, histograms, power spectra, bar charts, error charts, scatter plots, etc., with just a few lines of code.

# The Many (Inter)faces of Matplotlib

## **Pyplot**

- Provides a Matlab-style state-machine interface to the underlying object-oriented plotting library in matplotlib.
- Preferred method of access for interactive plotting.
- [http://matplotlib.sourceforge.net/users/pyplot\\_tutorial.html](http://matplotlib.sourceforge.net/users/pyplot_tutorial.html)

## **Pylab**

- Combines the pyplot functionality (for plotting) with the Numpy functionality (for mathematics and for working with arrays) in a single namespace, making that namespace (or environment) even more Matlab-like.
- Formerly preferred method of access for interactive plotting, but still available.

## **matplotlibAPI** (object-oriented)

- Used to embed matplotlib within an application, e.g. GFM
- <http://matplotlib.sourceforge.net/api>

# pyplot Vs. pylab

**pyplot:**

```
import matplotlib.pyplot as plt  
import numpy as np
```

```
x = np.arange(0, 10, 0.2)  
y = np.sin(x)
```

```
plt.plot(x, y)
```

```
plt.show()
```

**pylab:**

```
from pylab import *
```

```
x = arange(0, 10, 0.2)  
y = sin(x)
```

```
plot(x, y)
```

```
show()
```

# Syntax for Plotting

```
#!/usr/bin/env python
import matplotlib.pyplot as plt

x = [...]          # define the points on the x-axis
y = [...]          # define the points on the y-axis

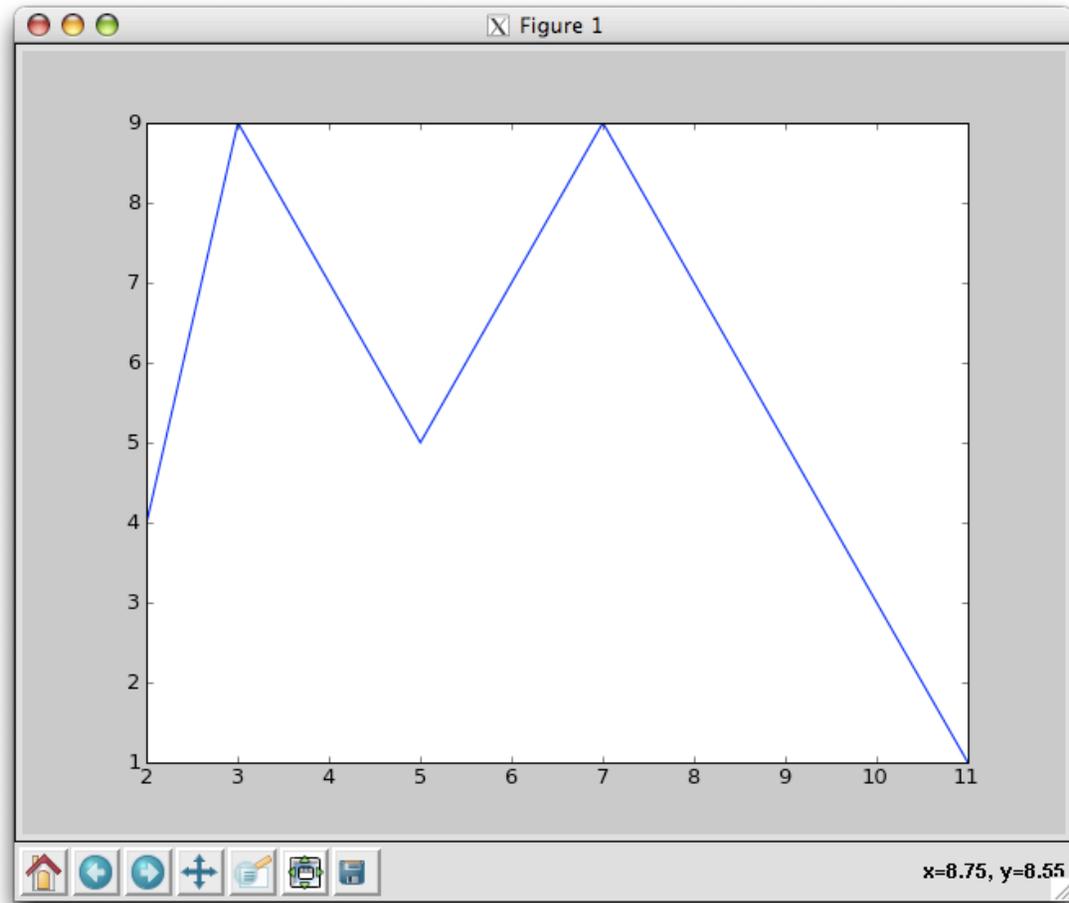
plt.plot(x,y)
plt.show()         # display the plot on the screen
```

# A Simple Example

```
#!/usr/bin/env python
import matplotlib.pyplot as plt

x = [2, 3, 5, 7, 11]
y = [4, 9, 5, 9, 1]
plt.plot(x, y)
plt.show()
```

# Basic Graph



# Some Pyplot Functions

```
plot(x,y)
xlabel('string')          # label the x-axis
ylabel('string')         # label the y-axis
title('string')          # write the title of the plot
grid(true/false)        # adds grid boxes
savefig('fileName.type') # type can be png, ps, pdf, etc
show()                   # display the graph on the screen
xlim(xmin,xmax)         # set/get the xlimits
ylim(ymin,ymax)         # set/get the ylimits
```

# Example

```
#!/usr/bin/env python
import math
import numpy as np
import matplotlib.pyplot as plt

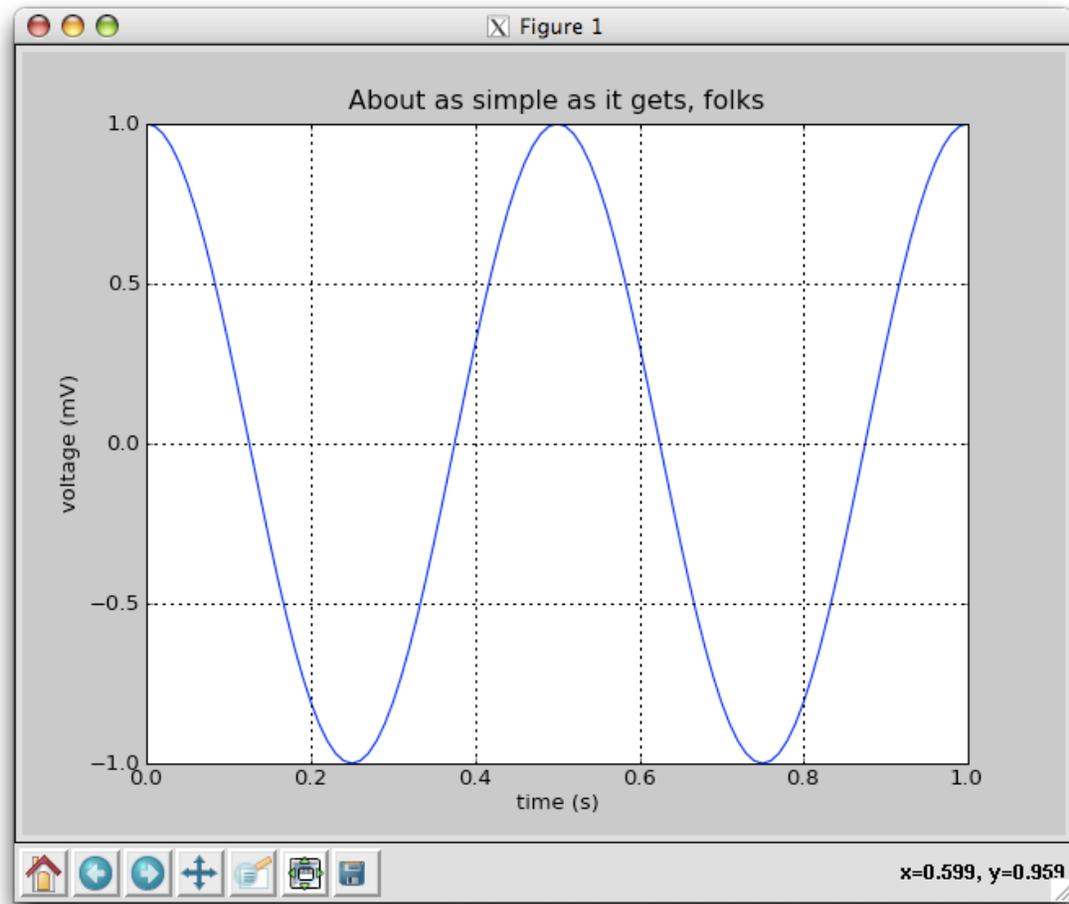
t = np.arange(0.0, 1.0+0.01, 0.01)
s = np.cos(2*2*math.pi*t)
plt.plot(t, s)

plt.xlabel('time (s)')
plt.ylabel('voltage (mV)')
plt.title('About as simple as it gets, folks')
plt.grid(True)
plt.savefig('simple_plot')

plt.show()

# file simplePlot.py
```

# Simple cosine Plot



# Multiples Figures and Axes (1)

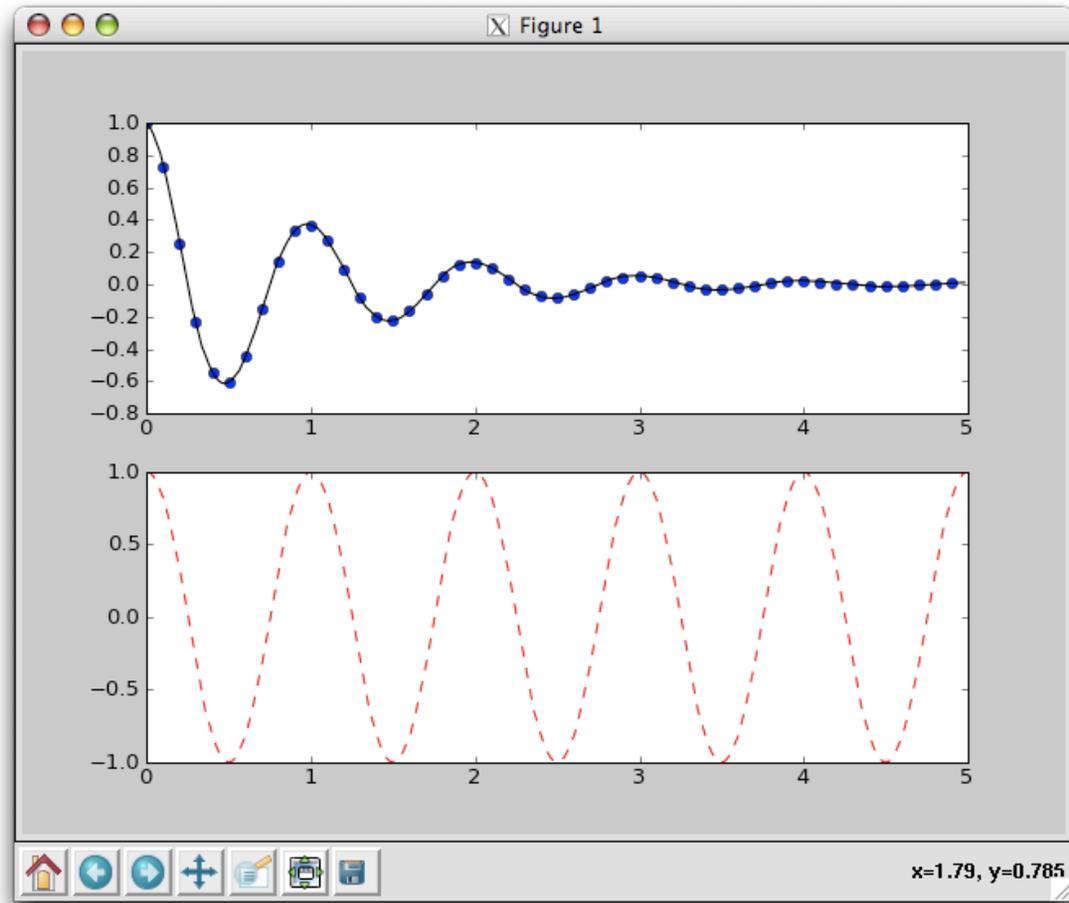
```
import numpy as np
import matplotlib.pyplot as plt
def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure(1)
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
# file multiplefiguresAxes_1.py
```

# Picture



## Multiples Figures and Axes (2)

`figure(num)`

# allows to plot multiple figures at the same time

# can be called several times

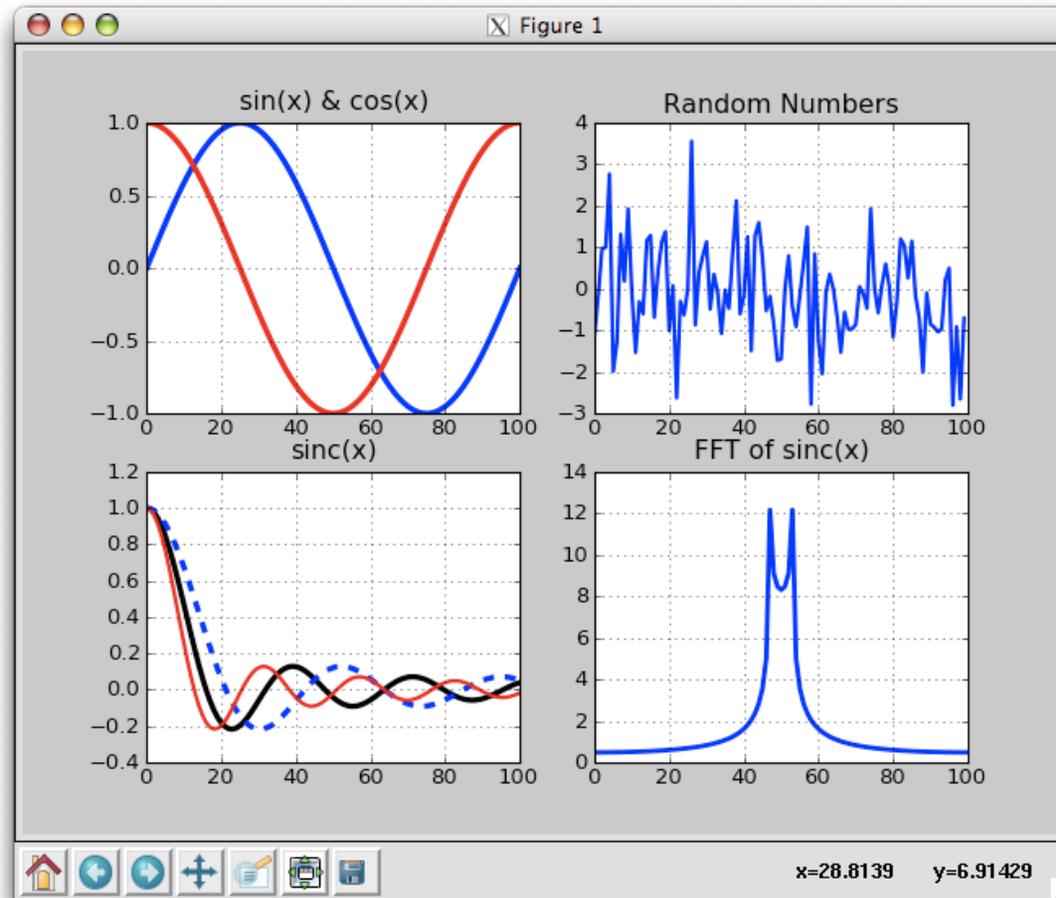
# num: reference number to keep track of the figure object

`subplot(numrows, numcols, fignum)`

# fignum range from  $\text{numrows} * \text{numcols}$

# `subplot(211)` is identical to `subplot(2,1,1)`

# Picture



## Multiples Figures and Axes (3)

```
import matplotlib.pyplot as plt
```

```
plt.figure(1)           # the first figure
plt.subplot(211)        # the first subplot in the first figure
plt.plot([1,2,3])
plt.subplot(212)        # the second subplot in the first figure
plt.plot([4,5,6])

plt.figure(2)           # a second figure
plt.plot([4,5,6])       # creates a subplot(111) by default

plt.figure(1)           # figure 1 current; subplot(212) still current
plt.subplot(211)        # make subplot(211) in figure1 current
plt.title('Easy as 1,2,3') # subplot 211 title

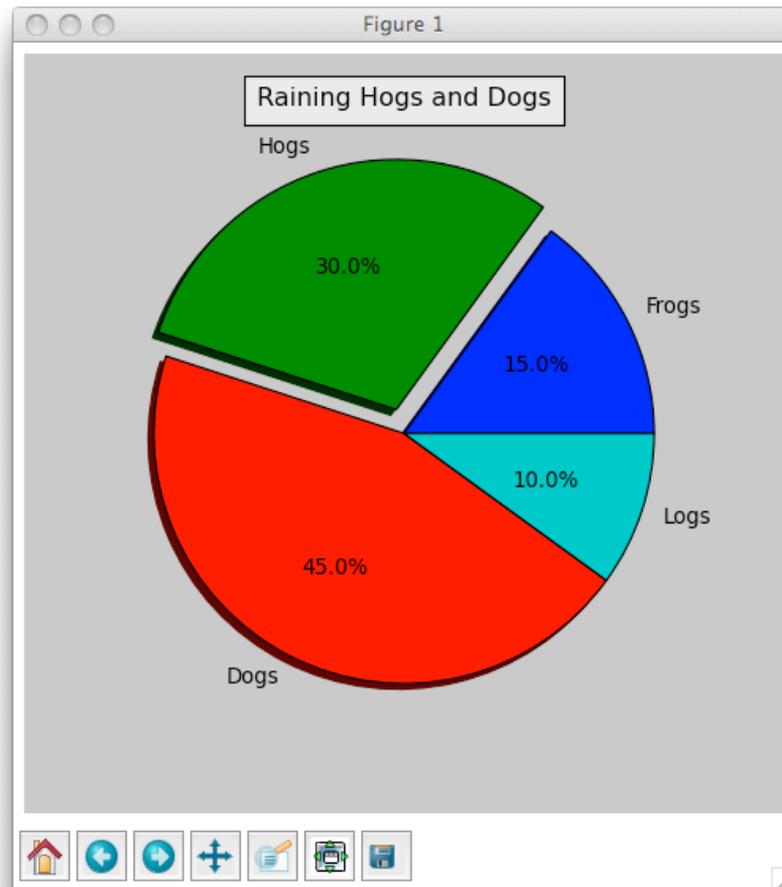
plt.show()

# file multipleFigureAxes_2.py
```

## Pie Chart

```
figure(1, figsize=(6,6))  
ax = axes([0.1, 0.1, 0.8, 0.8])  
  
labels = 'frogs', 'Hogs', 'Dogs', 'Logs'  
fracs = [15, 30, 45, 10]  
  
explode=(0, 0.05, 0, 0)  
  
pie(frac, explode=explode, labels=labels)
```

# Figure of a Pie Chart



# Histogram

```
import numpy as np
import matplotlib.pyplot as plt

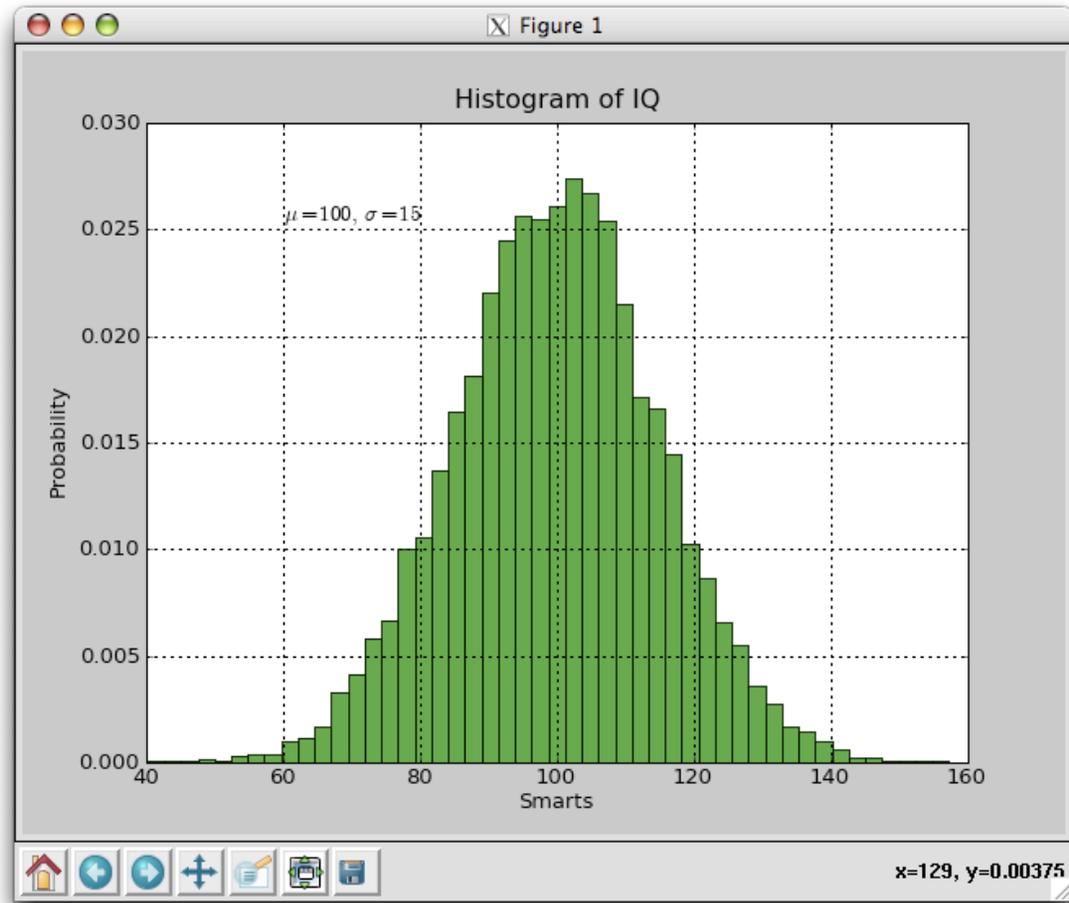
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g', alpha=0.75)

plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)

# file histogram.py
```

# Picture



# Using Mathematical Expressions in Text

- Matplotlib accepts TeX equation expressions in any text.
- Matplotlib has a built-in TeX parser
- To write the expression  $\sigma_i = 15$  in the title, you can write:

```
plt.title(r'$\sigma_i=15$')
```

where `r` signifies that the string is a raw string and not to treat backslashes and python escapes.

# Annotating Text

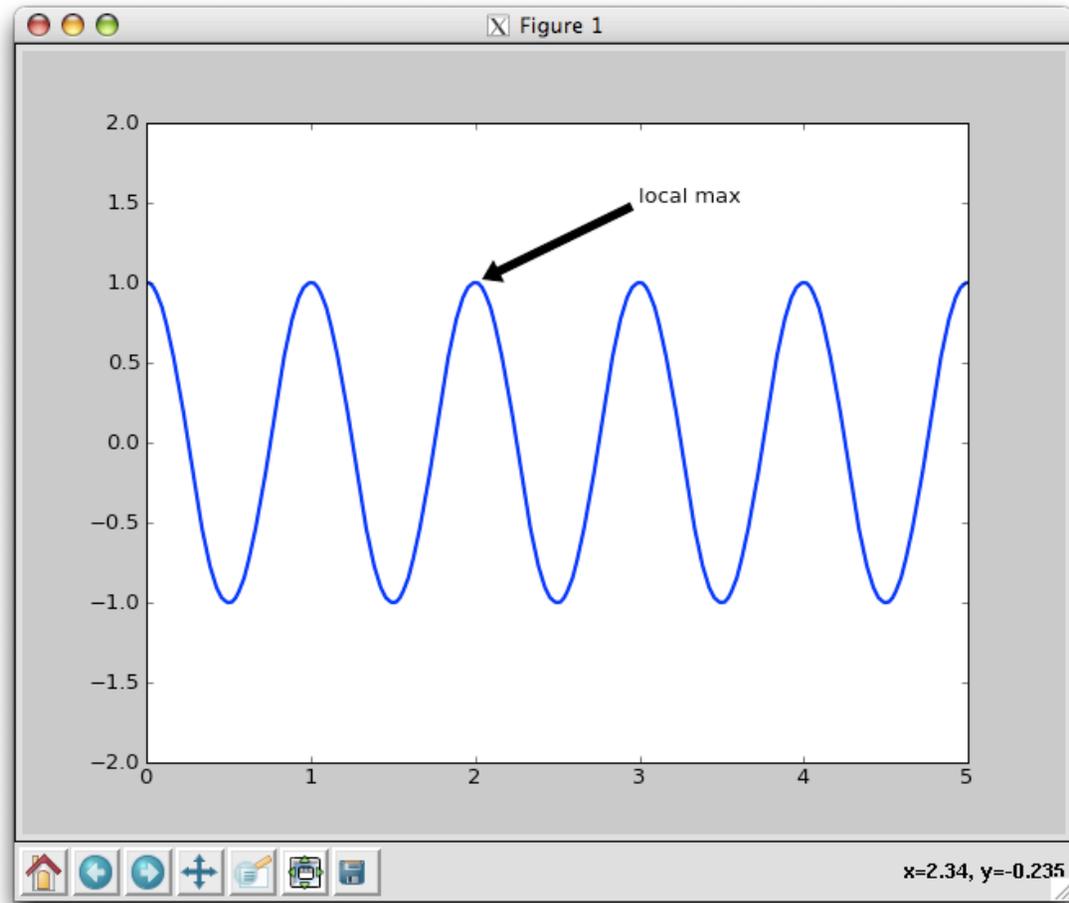
```
import numpy as np
import matplotlib.pyplot as plt

ax = plt.subplot(111)
t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = plt.plot(t, s, lw=2)

plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5), arrowprops=dict(facecolor='black',
    shrink=0.05),
    )

plt.ylim(-2,2)
plt.show()
```

# Picture



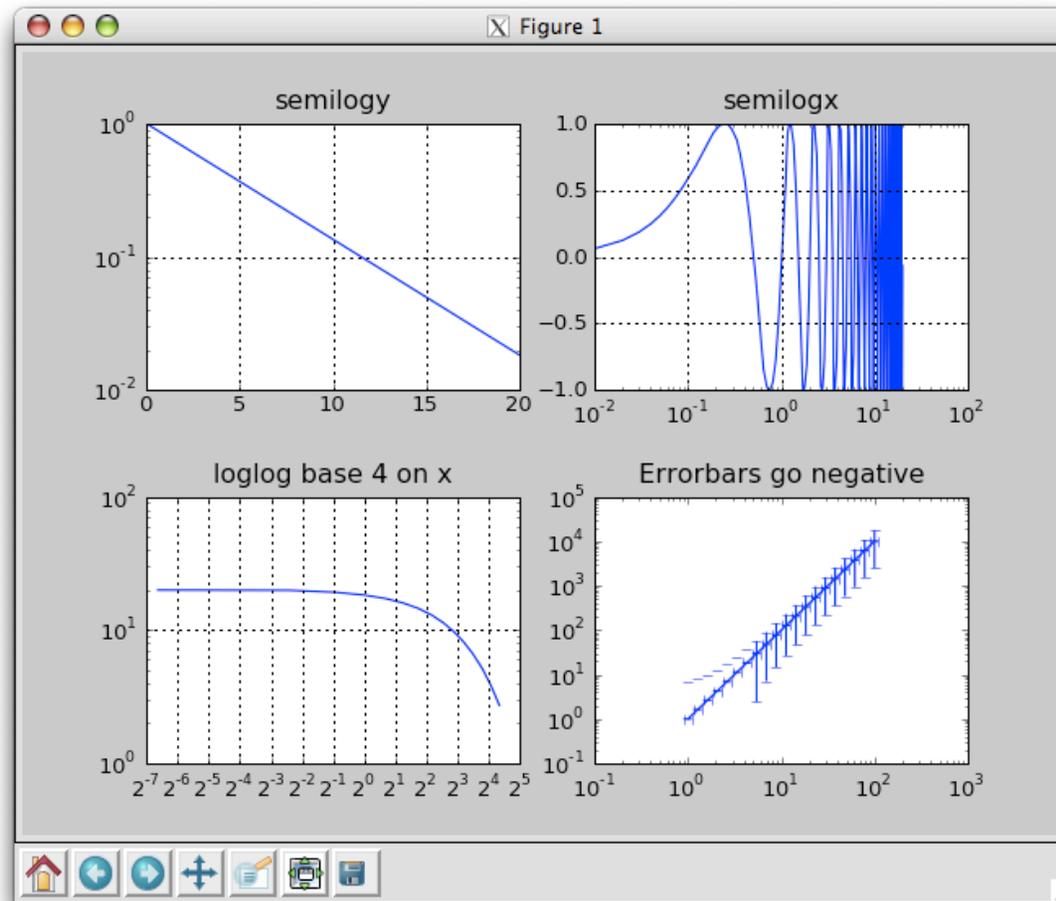
# Log Plots

Use the following pyplot functions:

- **semilogx():** make a plot with log scaling on the x axis
- **semilogy():** make a plot with log scaling on the y axis
- **loglog():** make a plot with log scaling on the x and y axis

**# file logPlots.py**

# Picture



## Plot with Fill

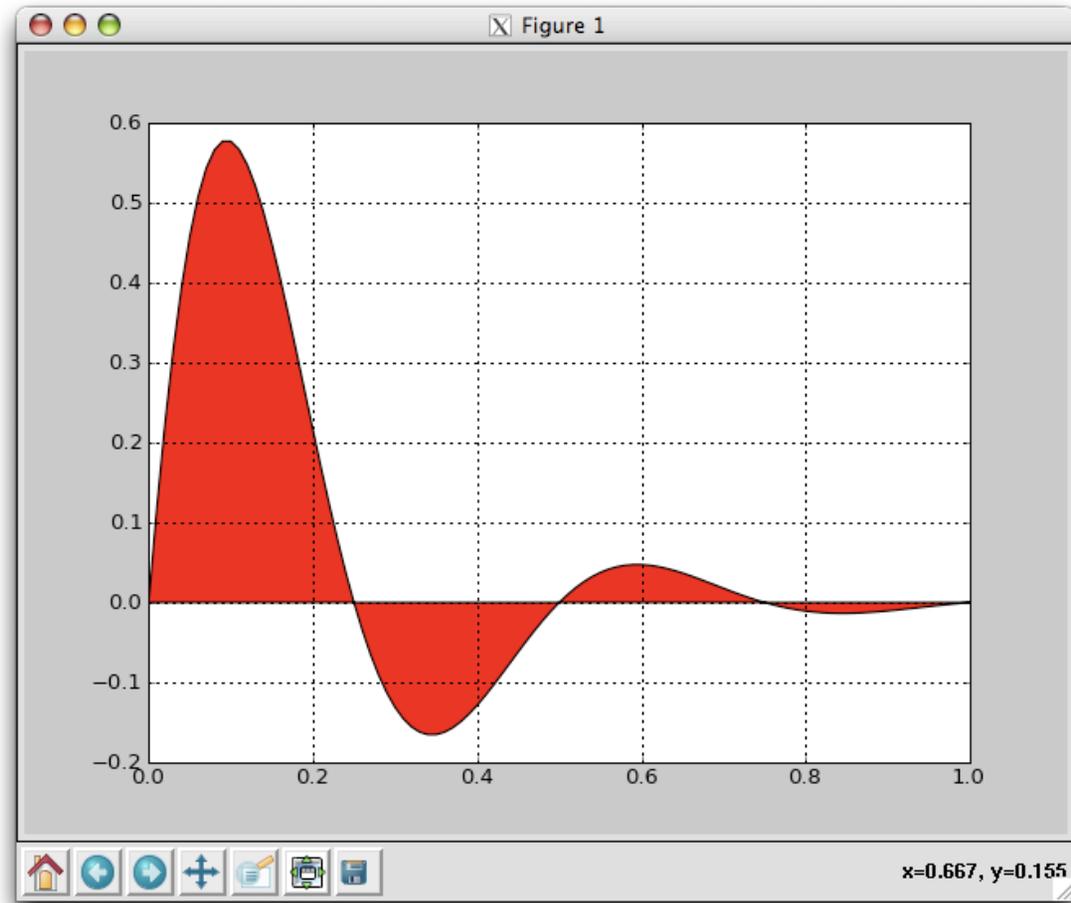
```
import numpy as np
import matplotlib.pyplot as plt

t = np.arange(0.0, 1.01, 0.01)
s = np.sin(2*2*np.pi*t)

plt.fill(t, s*np.exp(-5*t), 'r')
plt.grid(True)
plt.show()

# file fillPlot.py
```

# Picture



# Legend

Call signature:

**`legend(*args, **kwargs)`**

- Place a legend on the current axes at location *loc*.
- Labels are a sequence of strings
- *loc* can be a string or an integer

# Sample Legend Commands

**# make a legend with existing lines**

**legend()**

**# automatically generate the legend from labels**

**legend( 'label1', 'label2', 'label3' ) # automatically generate the legend from labels**

**# Make a legend for a list of lines and labels**

**legend( (line1, line2, line3), ('label1', 'label2', 'label3') )**

**# make a legend at a given location, using a location argument**

**legend( 'label1', 'label2', 'label3'), loc='upper left')**

**legend( (line1, line2, line3), ('label1', 'label2', 'label3'), loc=2)**

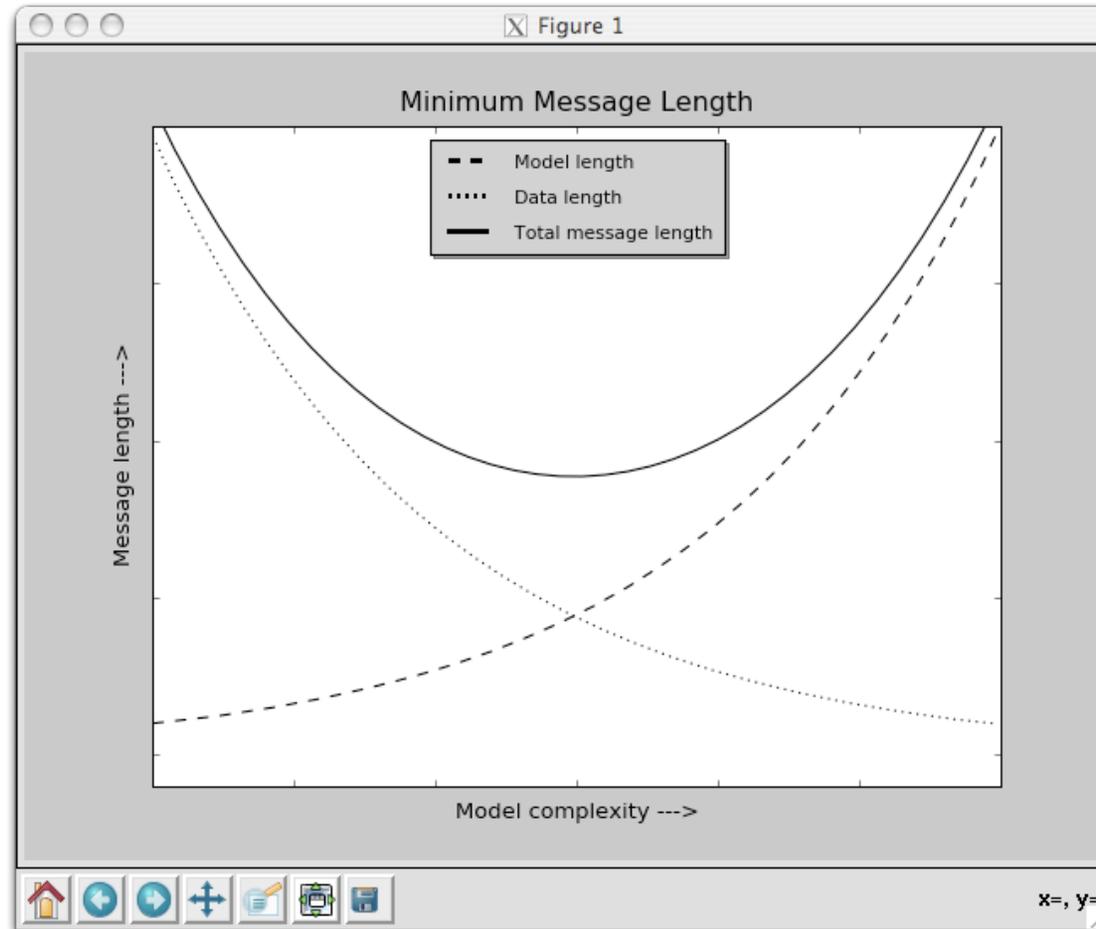
# Sample Codes with Legends

Check the files:

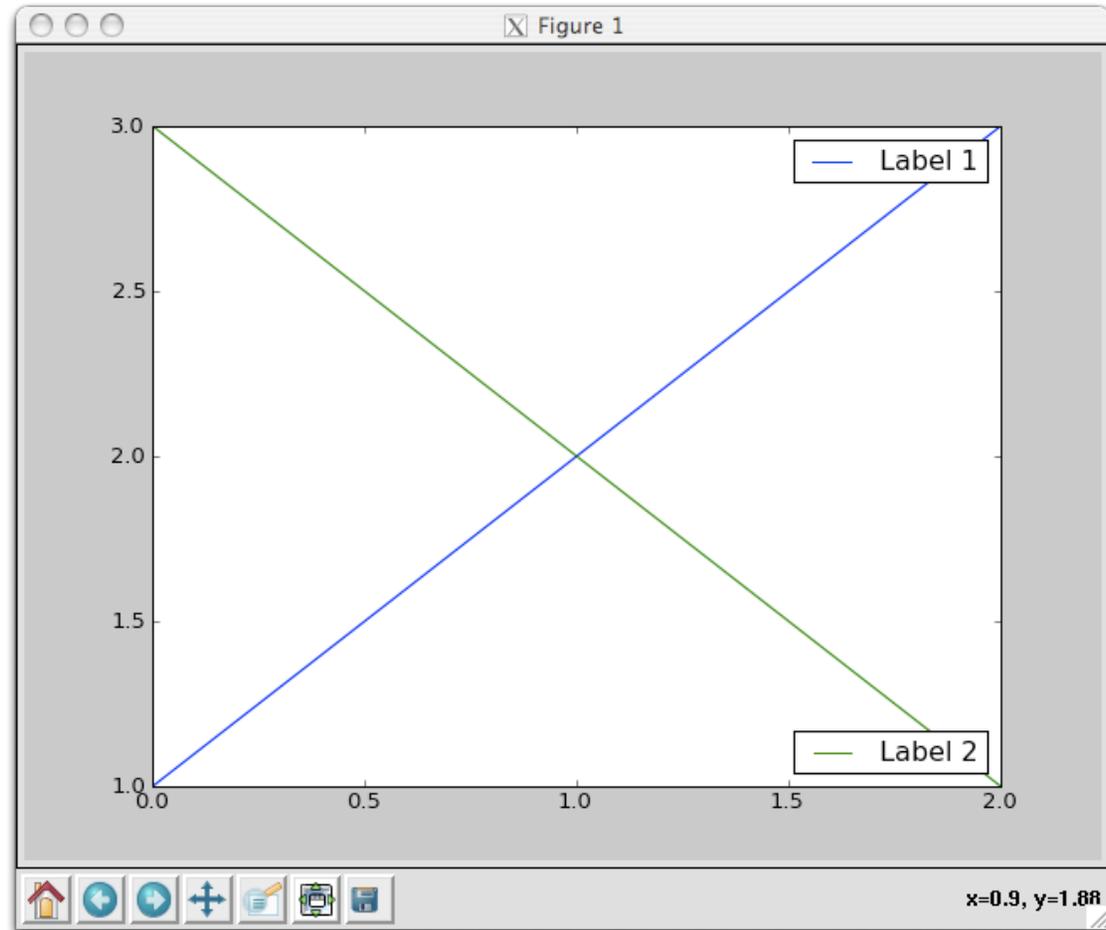
`demoLegend.py`

`multipleLegend.py`

# Picture



# Picture



# Colorbar

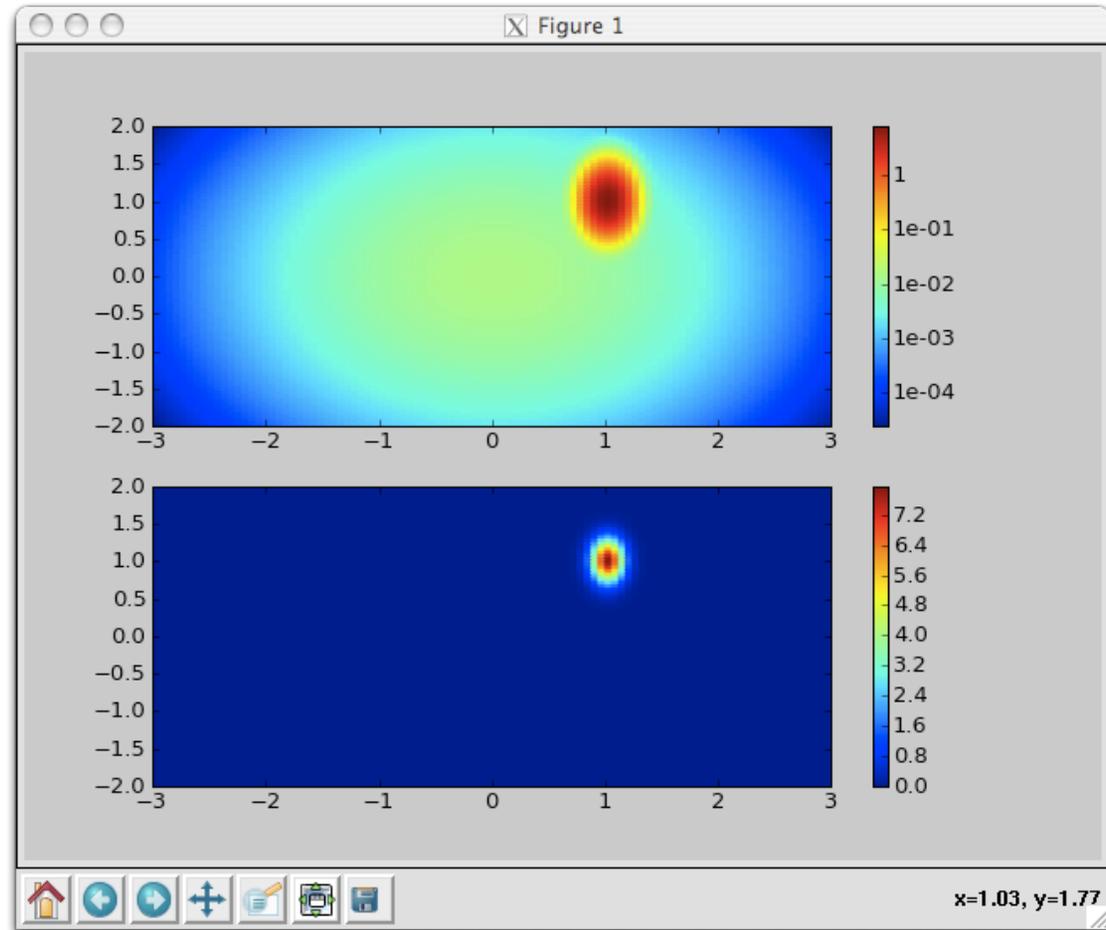
Add a colorbar to a plot. You only need to include the call:

```
colorbar()
```

Check the file:

```
sampleColorbar.py
```

# Picture



# Contour Plots

# Make a contour plot of an array Z. The level values are chosen automatically  
`contour(Z)`

# X, Y specify the (x, y) coordinates of the surface  
`contour(X, Y, Z)`

# contour N automatically-chosen levels  
`contour(Z,N)`  
`contour(X,Y,Z,N)`

# draw contour lines at the values specified in sequence V  
`contour(Z,V)`  
`contour(X,Y,Z,V)`

# fill the  $(\text{len}(V)-1)$  regions between the values in V  
`contourf(..., V)`

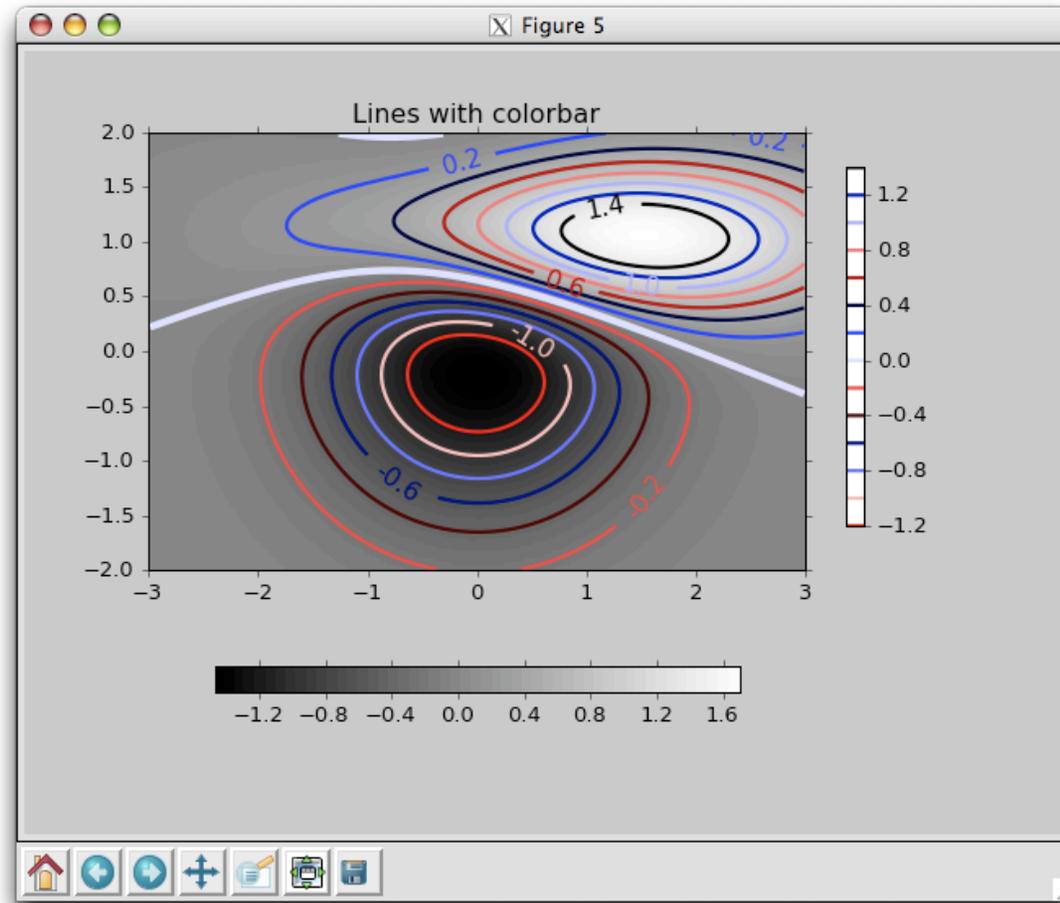
# Sample Contour Plots

Check the files:

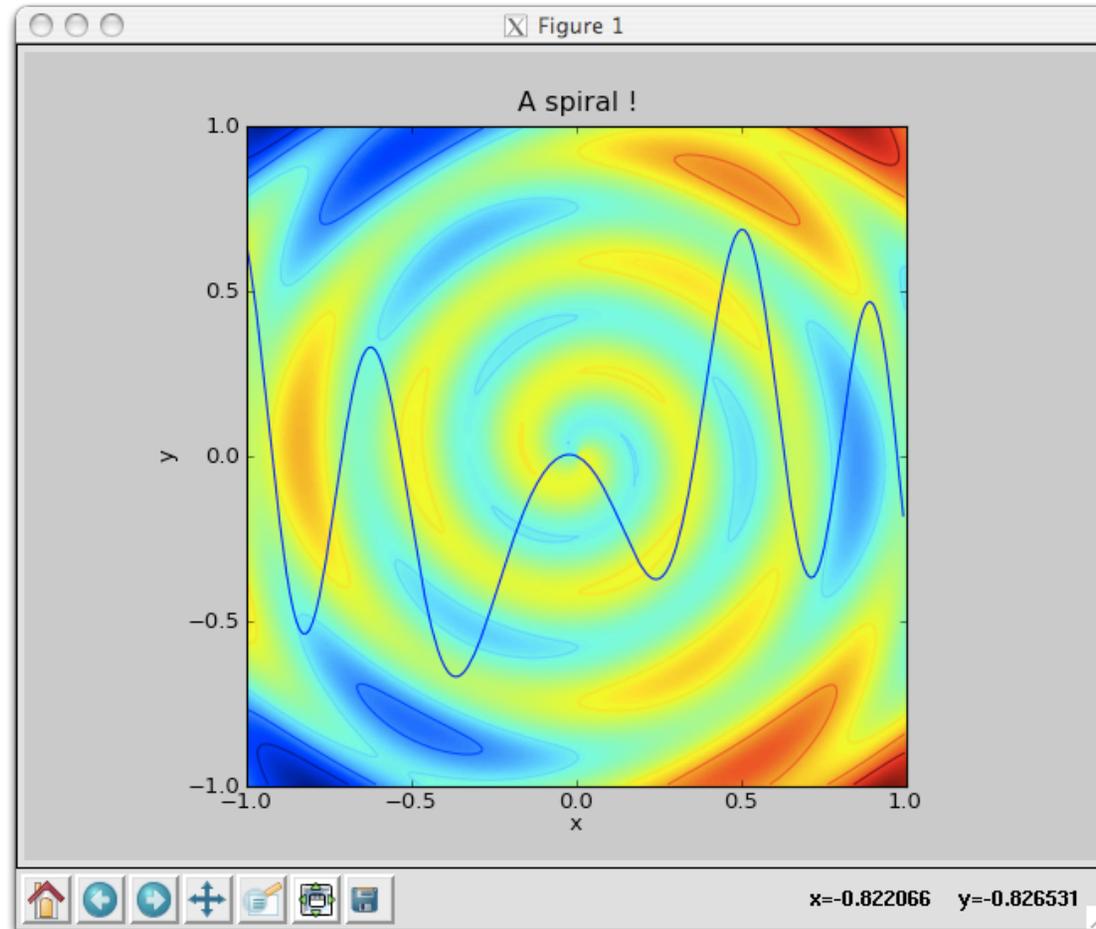
**demoContour.py**

**spiralPlotting.py**

# Picture



# Spiral Plot



# mplot3d

- The mplot3d toolkit adds simple 3d plotting capabilities to Matplotlib by supplying an axis object that can create a 2d projection of a 3d scene.
- It produces a list of 2d lines and patches that are drawn by the normal Matplotlib code.
- The resulting graph will have the same look and feel as regular 2d plots.
- Provide the ability to rotate and zoom the 3d scene.

# 3D Graphs

matplotlib's 3D capabilities were added by incorporating:

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
```

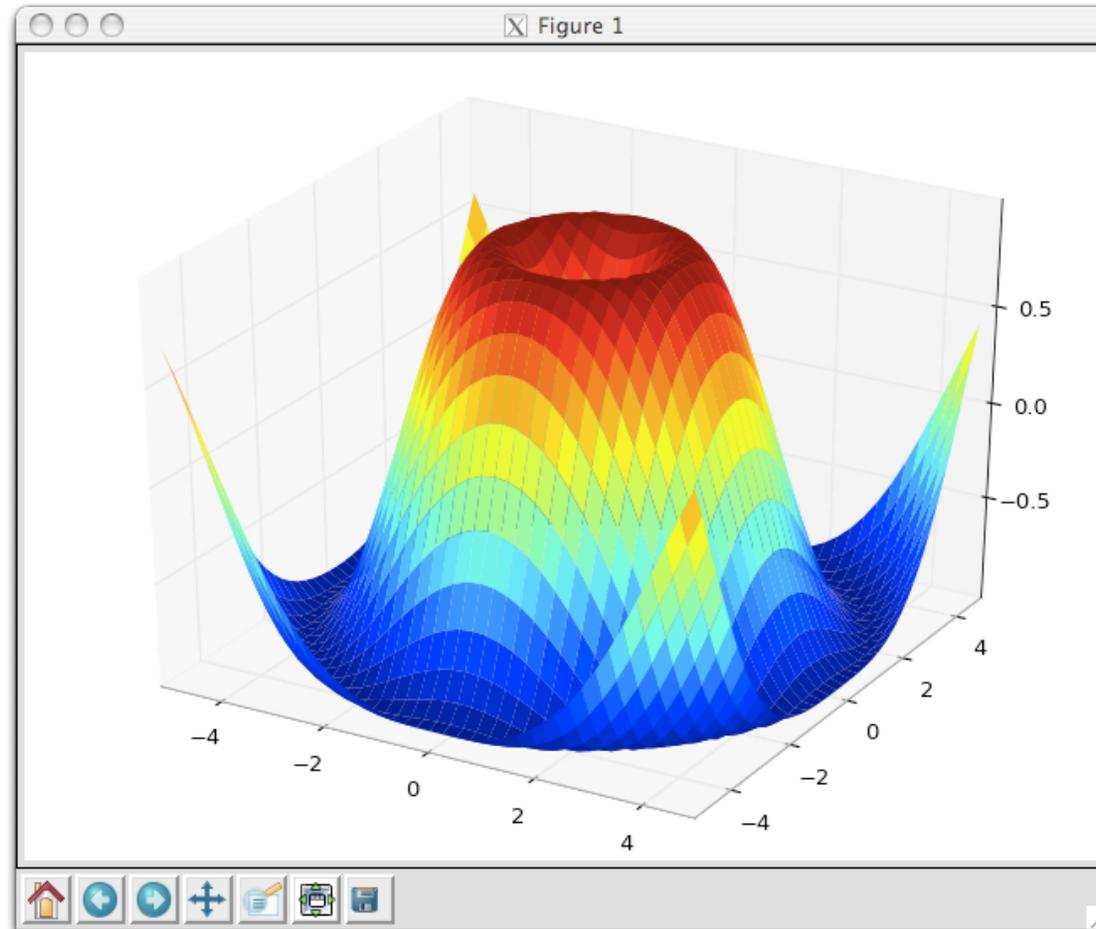
```
fig = plt.figure()
ax = Axes3D(fig)
x = ...
y = ...
z = ...
ax.plot(x,y,z, ...)
```

## Example

$$z = \sin\left(\sqrt{x^2 + y^2}\right)$$

$$-5 \leq x, y \leq 5$$

# 3D Surface Demo



# 2D/3D Plot Exercise

Consider the problem:

$$z = e^{-(x^2+y^2)} + 0.6e^{-((x+1.8)^2+y^2)}$$
$$-2 \leq x, y \leq 2$$

We want to plot the following (**sample2D3DGraphics.py**):

- 2D simple contour plot
- 2D contour plot with color filling spaces and color bar
- 3D wire frame plot
- 3D surface plot
- 3D filled contour plot

# Manipulating Images

```
from pylab import imread, imshow
```

```
a = imread('myImage.png')
```

```
imshow(a)
```

# Plotting Geographical Data using Basemap

- Matplotlib toolkit
- Collection of application-specific functions that extends Matplotlib functionalities
- Provides an efficient way to draw Matplotlib plots over real world maps
- Useful for scientists such as oceanographers and meteorologists.

# Defining a Basemap Object

```
import matplotlib.pyplot as plt          # pyplot module import
from mpl_toolkits.basemap import Basemap # basemap import
import numpy as np                       # Numpy import

# Lambert Conformal map of USA lower 48 states
m = Basemap(llcrnrlon=-119,
            llcrnrlat=22,
            urcrnrlon=-64,
            urcrnrlat=49,
            projection='lcc',
            lat_1=33,
            lat_2=45,
            lon_0=-95,
            resolution='h',
            area_thresh=10000)
```

# Comments

<b>projection</b>	Type of map projection used
<b>lat_1</b>	First standard parallel for lambert conformal, albers equal area and equidistant conic
<b>lat_2</b>	Second standard parallel for lambert conformal, albers equal area and equidistant conic.
<b>lon_0</b>	Central meridian (x-axis origin) - used by all projections
<b>llcrnrlon</b>	Longitude of lower-left corner of the desired map domain
<b>llcrnrlat</b>	Latitude of lower-left corner of the desired map domain
<b>urcrnrlon</b>	Longitude of upper-right corner of the desired map domain
<b>urcrnrlat</b>	Latitude of upper-right corner of the desired map domain
<b>resolution</b>	Specifies what the resolution is of the features added to the map (such as coast lines, borders, and so on), here we have chosen high resolution (h), but crude, low, and intermediate are also available.
<b>area_thresh</b>	Specifies what the minimum size is for a feature to be plotted. In this case, only features bigger than 10,000 square kilometer

# Defining Borders

```
# draw the coastlines of continental area  
m.drawcoastlines()
```

```
# draw country boundaries  
m.drawcountries(linewidth=2)
```

```
# draw states boundaries (America only)  
m.drawstates()
```

## Coloring the Map

```
# fill the background (the oceans)  
m.drawmapboundary(fill_color='aqua')
```

```
# fill the continental area  
# we color the lakes like the oceans  
m.fillcontinents(color='coral',lake_color='aqua')
```

## Drawing Parallels & Meridians

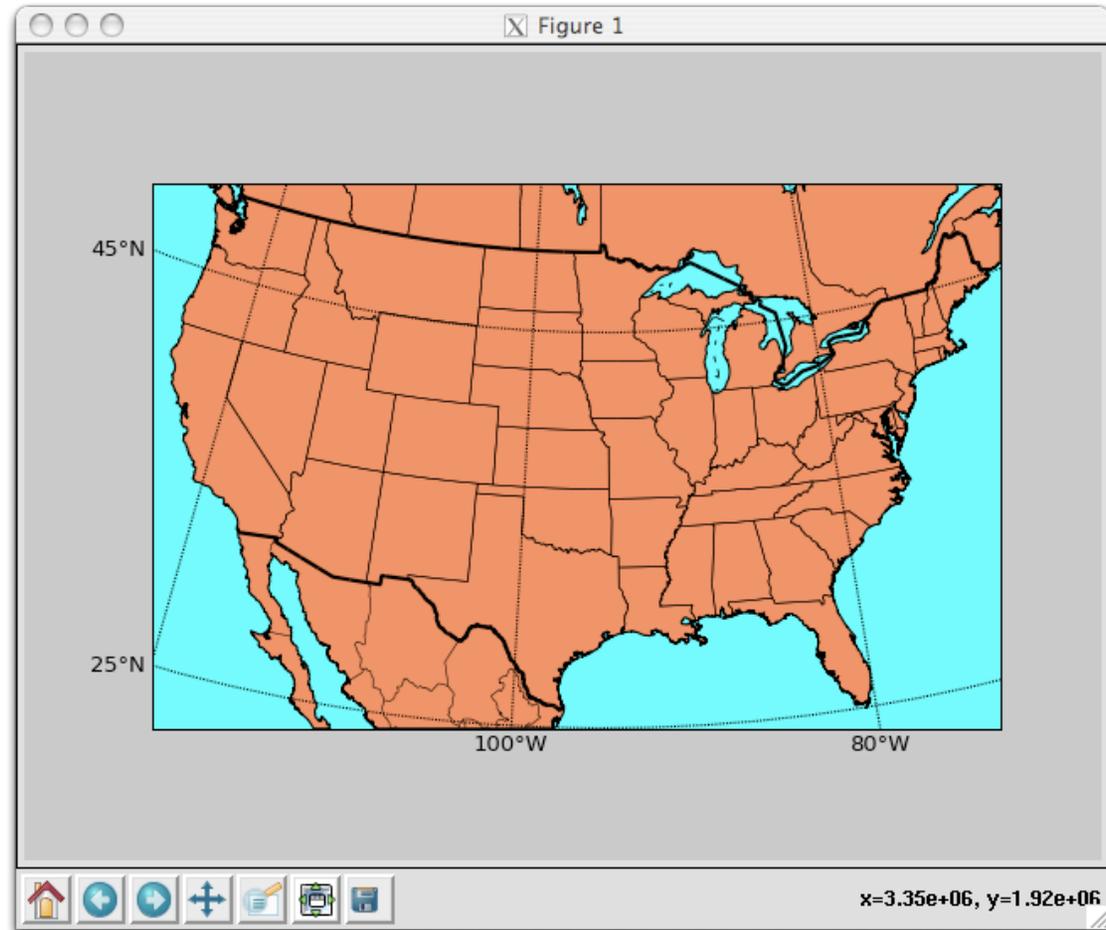
```
# We draw a 20 degrees graticule of parallels and  
# meridians for the map. Note how the labels argument  
# controls the positions where the graticules are labeled  
# labels=[left, right, top, bottom]
```

```
m.drawparallels(np.arange(25,65,20),labels=[1,0,0,0])
```

```
m.drawmeridians(np.arange(-120,-40,20),labels=[0,0,0,1])
```

```
# see file basemapSample.py
```

# Picture

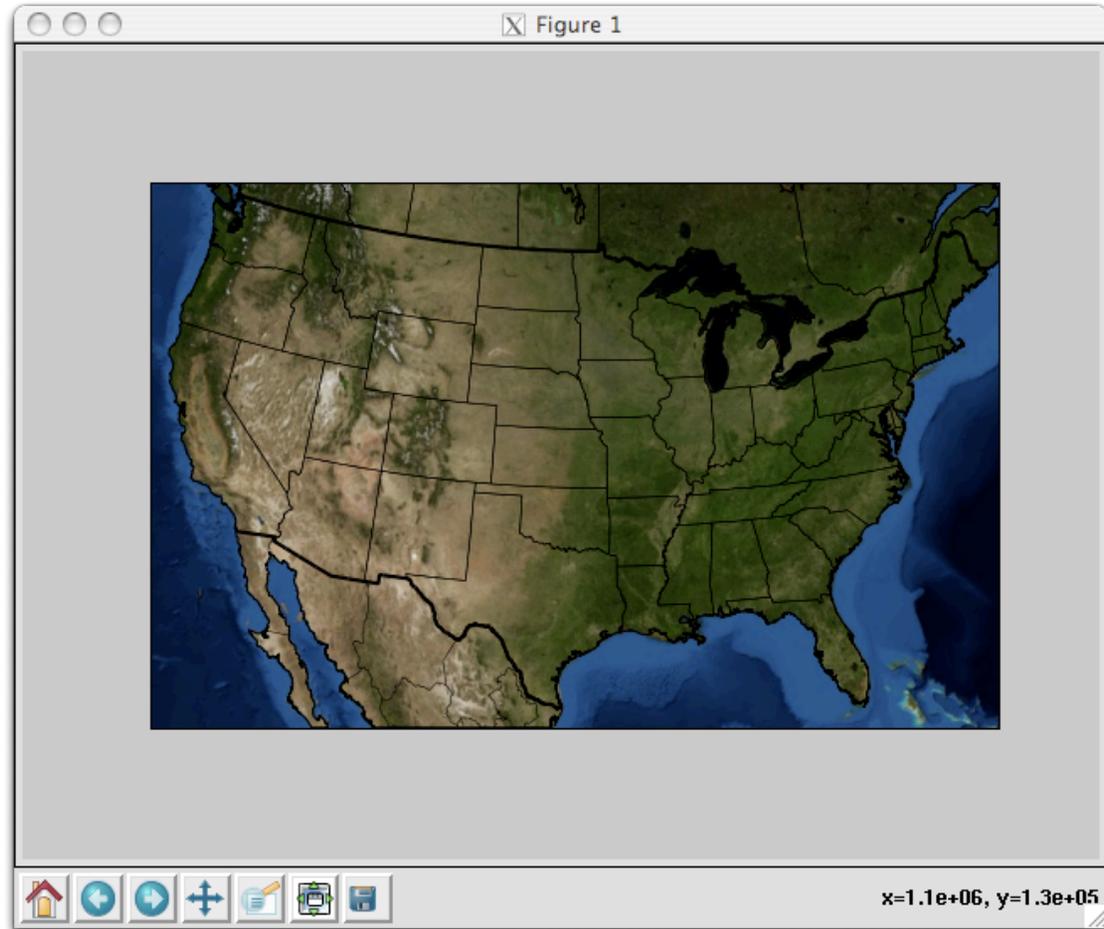


# Using Satellite Background

```
# display blue marble image (from NASA) as map background  
m.bluemarble()
```

```
# see file basemapSatellite.py
```

# Picture



# Plotting Data over a Map (1)

```
# Cities names and coordinates
cities = ['London', 'New York', 'Madrid', 'Cairo', 'Moscow', 'Delhi', 'Dakar']
lat = [51.507778, 40.716667, 40.4, 30.058, 55.751667, 28.61, 14.692778]
lon = [-0.128056, -74, -3.683333, 31.229, 37.617778, 77.23, -17.446667]

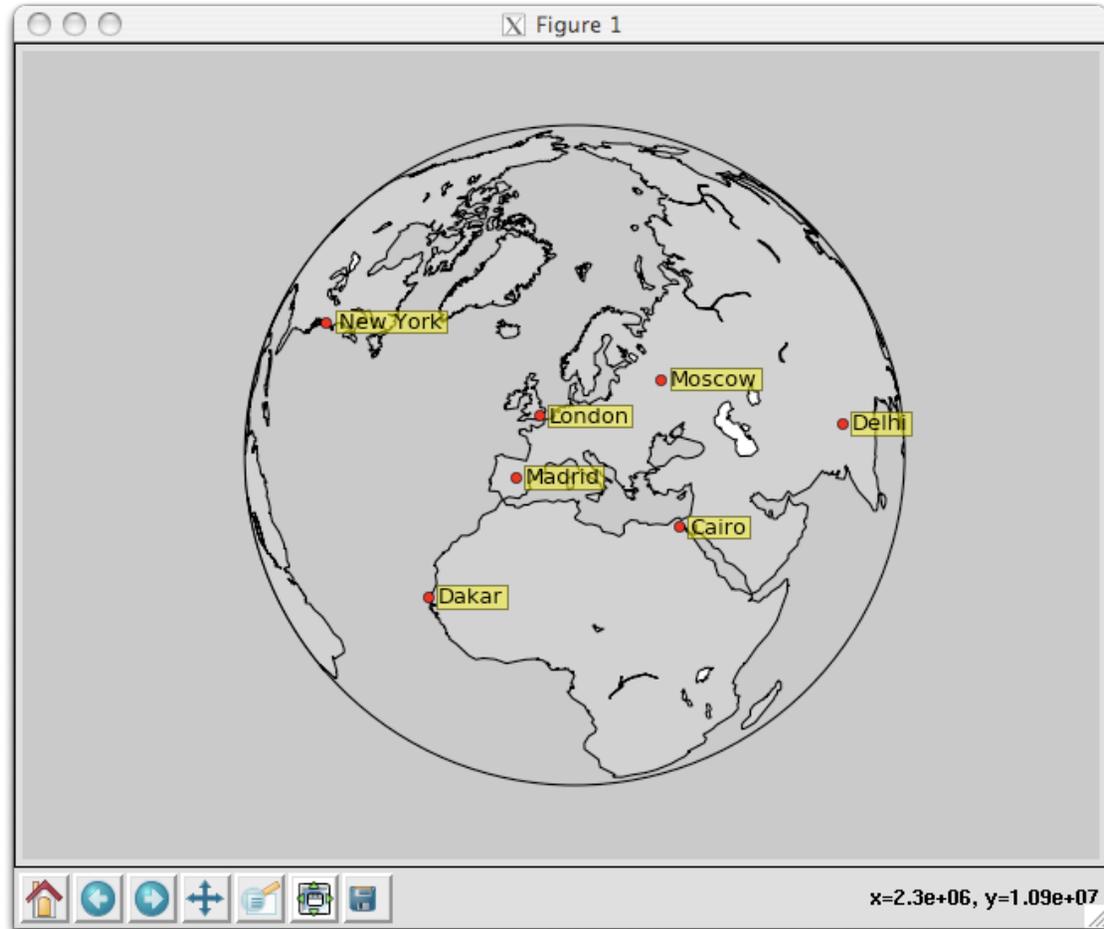
# orthogonal projection of the Earth
m = Basemap(projection='ortho', lat_0=45, lon_0=10)

# map city coordinates to map coordinates
x, y = m(lon, lat)

# draw a red dot at cities coordinates
plt.plot(x, y, 'ro')

# see file basemapDataPlot_1.py
```

# Picture

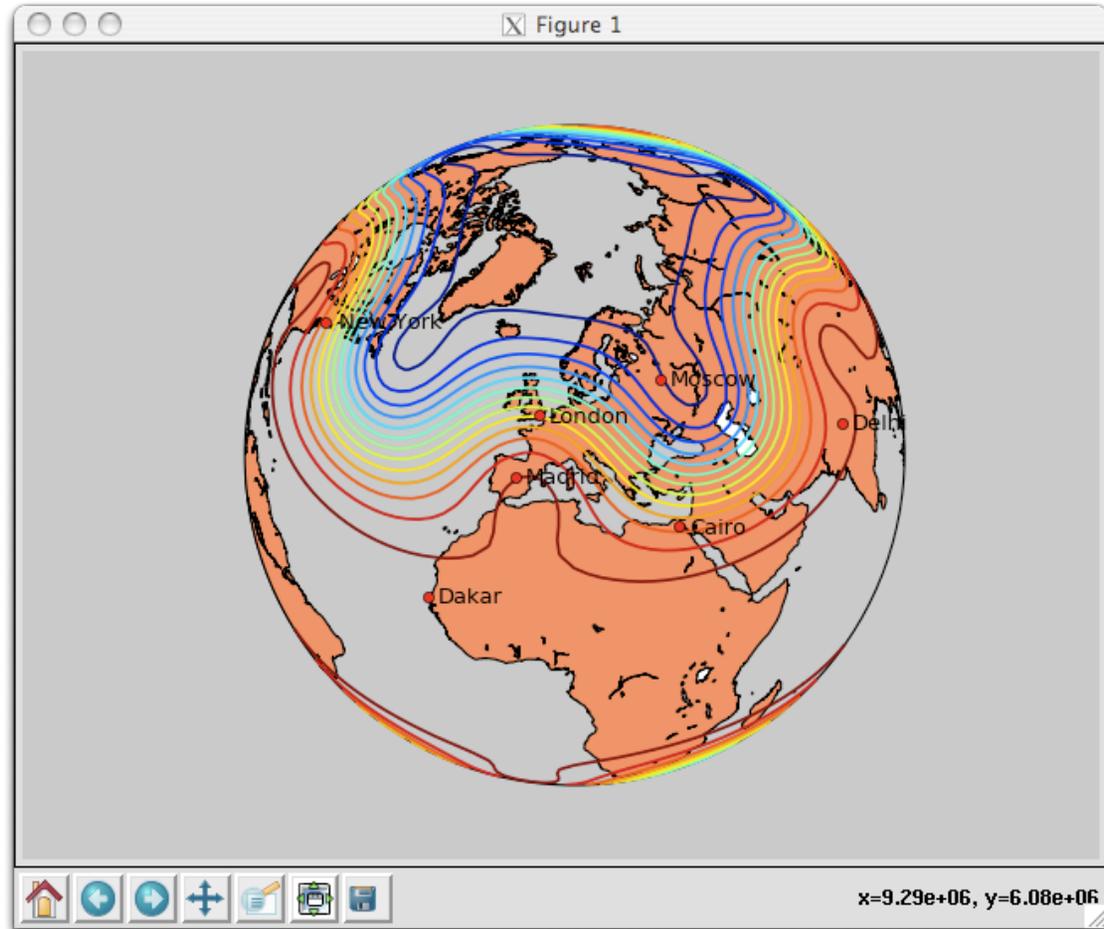


## Plotting Data over a Map (2)

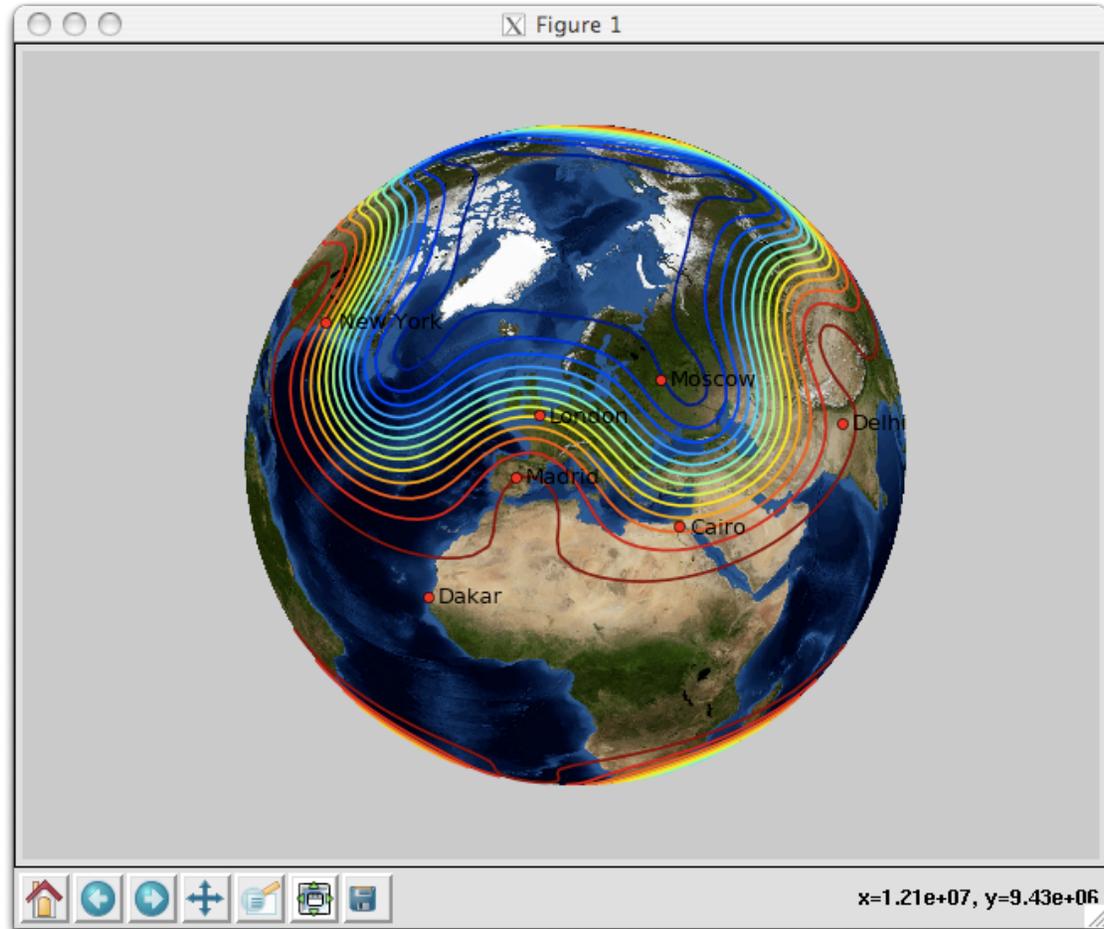
```
# make up some data on a regular lat/lon grid.
nlats = 73; nlons = 145; delta = 2.*np.pi/(nlons-1)
lats = (0.5*np.pi-delta*np.indices((nlats,nlons))[0,:,:])
lons = (delta*np.indices((nlats,nlons))[1,:,:])
wave = 0.75*(np.sin(2.*lats)**8*np.cos(4.*lons))
mean = 0.5*np.cos(2.*lats)*((np.sin(2.*lats))**2 + 2.)
# compute native map projection coordinates of lat/lon grid.
x, y = m(lons*180./np.pi, lats*180./np.pi)
# contour data over the map.
CS = m.contour(x,y,wave+mean,15,linewidths=1.5)

# See file basemapDataPlot_2.py
```

# Picture



# Picture



# netCDF4

- **Creating/Opening a file**
- **Dimension**
- **Variables**
- **Attributes**
- **Writing/Retrieving Data**

# Creating/Opening a netCDF File

```
from netCDF4 import Dataset
ncFid = Dataset(ncFileName, mode=modeType,
               format=fileFormat)
print ncFid.file_format
ncFid.close()
```

**modeType** can be: 'w', 'r+', 'r', or 'a'

**fileFormat** can be: 'NETCDF3\_CLASSIC', 'NETCDF3\_64BIT', 'NETCDF4\_CLASSIC',  
'NETCDF4'

## Dimension is a netCDF File

```
time = ncFid.createDimension('time', None)
```

```
lev = ncFid.createDimension('lev', 72)
```

```
lat = ncFid.createDimension('lat', 91)
```

```
lon = ncFid.createDimension('lon', 144)
```

```
print ncFid.dimensions
```

## Variables in a netCDF File

```
times = ncFid.createVariable('time','f8',('time',))
```

```
levels = ncFid.createVariable('lev','i4',('lev',))
```

```
latitudes = ncFid.createVariable('lat','f4',('lat',))
```

```
longitudes = ncFid.createVariable('lon','f4',('lon',))
```

```
temp = ncFid.createVariable('temp','f4',('time','lev','lat','lon',))
```

## Attributes in netCDF File

ncFid.description = 'Sample netCDF file'

ncFid.history = 'Created for GISS on November 29, 2012'

ncFid.source = 'netCDF4 python tutorial'

latitudes.units = 'degrees north'

longitudes.units = 'degrees east'

levels.units = 'hPa'

temp.units = 'K'

times.units = 'hours since 0001-01-01 00:00:00.0'

times.calendar = 'gregorian'

# Writing Data

```
import numpy
latitudes[:] = numpy.arange(-90,91,2.0)
longitudes[:] = numpy.arange(-180,180,2.5)
levels[:] = numpy.arange(0,72,1)

from numpy.random import uniform
temp[0:5,:,:,:] = uniform(size=(5,levels.size,latitudes.size,
    longitudes.size))

# ncWriting.py
```

# Reading Data

```
ncFid = Dataset('myFile.nc4', mode='r')
```

```
time = ncFid.variables['time'][:]
```

```
lev = ncFid.variables['lev'][:]
```

```
lat = ncFid.variables['lat'][:]
```

```
lon = ncFid.variables['lon'][:]
```

```
temp = ncFid.variables['temp'][:]
```

```
# ncReading.py
```

# Python-based EOF computation

- Goals
- Data
- Verification
- Approaches
  - CDAT + eof2
  - R + Rpy2

# Goals

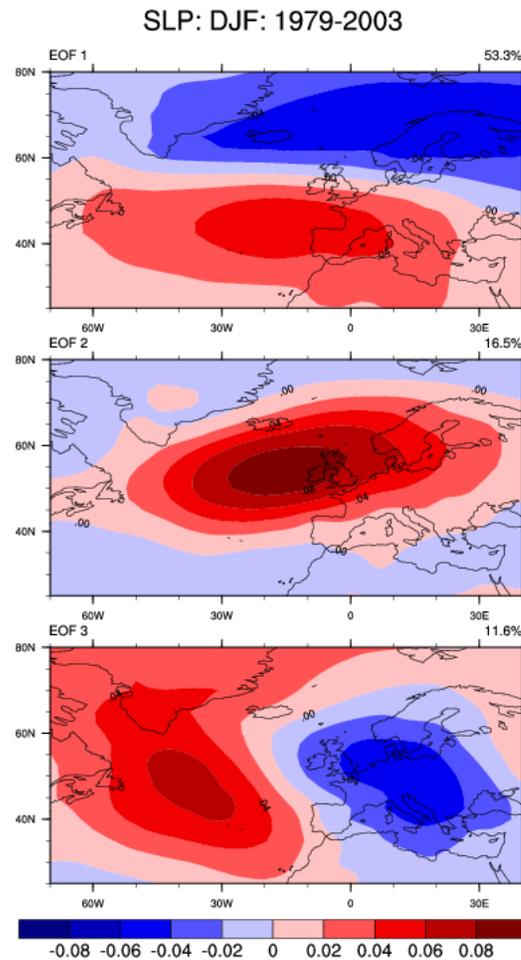
- Needs to work
- Needs to be reasonably easy/straightforward to use
- Best if it is maintained
- Easily installed

# Data

- Sea level pressure
- Source:
  - **NCEP/NCAR Reanalysis 1: Surface**
  - <http://www.esrl.noaa.gov/psd/data/gridded/data.ncep.reanalysis.surface.html>
  - [slp.mon.mean.nc](http://slp.mon.mean.nc)
    - Monthly means from 1948 to present
    - 2.5 degree latitude x 2.5 degree longitude global grid (144x73)
    - 90N - 90S, 0E - 357.5E

# Verification

- EOF example using NCAR Command Language (NCL)
  - Good demonstration of typical steps involved in EOF calculations (data sub-setting, seasonal time averaging, plotting of EOFs, etc.)
- Source:
  - <http://www.ncl.ucar.edu/Applications/eof.shtml>



# Verification

- NCL Source Code
  - [http://www.ncl.ucar.edu/Applications/Scripts/eof\\_1.ncl](http://www.ncl.ucar.edu/Applications/Scripts/eof_1.ncl)
  - Next page

```

; =====
; eof_1.ncl
;
; Concepts illustrated:
; - Calculating EOFs
; - Using coordinate subscripting to read a specified geographical region
; - Rearranging longitude data to span -180 to 180
; - Calculating symmetric contour intervals
; - Drawing filled bars above and below a given reference line
; - Drawing subtitles at the top of a plot
;
; =====
; Calculate EOFs of the Sea Level Pressure over the North Atlantic.
; =====
; The slp.mon.mean file can be downloaded from:
; http://www.esrl.noaa.gov/psd/data/gridded/data.ncep.reanalysis.surface.html
; =====
load "$NCARG_ROOT/lib/ncarg/nclscripts/csm/gsn_code.ncl"
load "$NCARG_ROOT/lib/ncarg/nclscripts/csm/gsn_csm.ncl"
load "$NCARG_ROOT/lib/ncarg/nclscripts/csm/contributed.ncl"

begin
; =====
; User defined parameters that specify region of globe and
; =====
lats = 25.
latN = 50.
lonL = -70.
lonR = 40.

yrStrt = 1979
yrLast = 2003

season = "DJF" ; choose Dec-Jan-Feb seasonal mean

neof = 3 ; number of EOFs
optEOF = True
optEOF@jopt = 0 ; This is the default; most commonly used; no need to specify.
;optEOF@jopt = 1 ; **only** if the correlation EOF is desired

optETS = False
; =====
; Open the file: Read only the user specified period
; =====
f = addfile ("slp.mon.mean.nc", "r")

TIME = f->time
YYYY = cd_calendar(TIME,-1)/100 ; entire file
iYYYY = ind(YYYY.ge.yrStrt .and. YYYY.le.yrLast)

slp = f->slp(iYYYY, :, :)
printVarSummary(slp) ; variable overview
; =====
; dataset longitudes span 0->357.5
; Because EOFs of the North Atlantic Oscillation are desired
; use the "lonFlip" (contributed.ncl) to reorder
; longitudes to span -180 to 177.5: facilitate coordinate subscripting
; =====
slp = lonFlip( slp )
printVarSummary(slp) ; note the longitude coord
; =====
; compute desired global seasonal mean: month_to_season (contributed.ncl)
; =====
SLP = month_to_season( slp, season )
nyrs = dimsizes(SLP@time)
printVarSummary(SLP)
; =====
; create weights: sqrt(cos(lat)) [or sqrt(gw) ]
; =====
rad = 4.*atan(1./180.
clat = f->lat
clat = sqrt( cos(rad*clat) ) ; gw for gaussian grid
; =====
; weight all observations
; =====
wSLP = SLP ; copy meta data
wSLP = SLP*conform(SLP, clat, 1)
wSLP@long_name = "Wgt: "+wSLP@long_name

; Reorder (lat,lon,time) the *weighted* input data
; Access the area of interest via coordinate subscripting
; =====
x = wSLP({lat|lats:latN},{lon|lonL:lonR},time{:})

eof = eofunc_Wrap(x, neof, optEOF)
eof_ts = eofunc_ts_Wrap(x, eof, optETS)

printVarSummary( eof ) ; examine EOF variables
printVarSummary( eof_ts )

; =====
; Normalize time series: Sum spatial weights over the area of used
; =====
dimx = dimsizes(x)
mln = dimx(1)
sumWgt = mln*sum( clat({lat|lats:latN}) )
eof_ts = eof_ts/sumWgt

; =====
; Extract the YYYYMM from the time coordinate
; associated with eof_ts [same as SLP@time]
; =====
yyyyymm = cd_calendar(eof_ts@time,-2)/100
; y:frac = yyyyymm_to_yyyyfrac(yyyyymm, 0.0); not used here
; =====
; PLOTS
; =====
wks = gsn_open_wks("ps", "eof")
gsn_define_colormap(wks, "BlWhRe") ; choose colormap
plot = new(neof,graphic) ; create graphic array
; only needed if paneling

; EOF patterns
res = True ; don't draw yet
res@gsnDraw = False ; don't advance frame yet
res@gsnFrame = False ; don't advance frame yet

;--This resource not needed in V6.1.0
res@gsnSpreadColors = True ; spread out color table
res@gsnAddCyclic = False ; plotted data are not cyclic

res@mpFillOn = False ; turn off map fill
res@mpMinLatP = lats ; zoom in on map
res@mpMaxLatP = latN
res@mpMinLonP = lonL
res@mpMaxLonP = lonR

res@cnFillOn = True ; turn on color fill
res@cnLinesOn = False ; True is default
res@cnLineLabelsOn = False ; True is default
res@lbLabelBarOn = False ; turn off individual lb's

symMinMaxPlt(eof, 16, False, res) ; set symmetric plot min/max
; contributed.ncl

; panel plot only resources
resP = True ; modify the panel plot
resP@gsnMaximize = True ; large format
resP@gsnPanelLabelBar = True ; add common colorbar
resP@lbLabelAutoStride = True ; auto stride on labels

yStrt = yyyyymm(0)/100
yLast = yyyyymm(nyrs-1)/100
resP@txString = "SLP: "+season+": "+yStrt+"-"+yLast

; =====
; first plot
; =====
do n=0,neof-1
res@gsnLeftString = "EOF "+(n+1)
res@gsnRightString = sprintf("%5.1f", eof@pcvar(n)) +"%"
plot(n)=gsn_csm_contour_map_ce(wks,eof(n, :, :),res)
end do
gsn_panel(wks,plot,(/neof,1/),resP) ; now draw as one plot
; =====
; second plot
; =====
; EOF time series [bar form]

rts = True
rts@gsnDraw = False ; don't draw yet
rts@gsnFrame = False ; don't advance frame yet
rts@gsnScale = True ; force text scaling

; these four rtsources allow the user to stretch the plot size, and
; decide exactly where on the page to draw it.

rts@vpHeightF = 0.40 ; Changes the aspect ratio
rts@vpWidthF = 0.85
rts@vpXF = 0.10 ; change start locations
rts@vpYF = 0.75 ; the plot

rts@tiYAxisString = "Pa" ; y-axis label

rts@gsnRefLine = 0. ; reference line
rts@gsnXYBarChart = True ; create bar chart
rts@gsnAboveRefLineColor = "red" ; above ref line fill red
rts@gsnBelowRefLineColor = "blue" ; below ref line fill blue

; panel plot only resources
rtsP = True ; modify the panel plot
rtsP@gsnMaximize = True ; large format
rtsP@txString = "SLP: "+season+": "+yStrt+"-"+yLast

year = yyyyymm/100

; create individual plots
do n=0,neof-1
rts@gsnLeftString = "EOF "+(n+1)
rts@gsnRightString = sprintf("%5.1f", eof@pcvar(n)) +"%"
plot(n) = gsn_csm_xy(wks,year,eof_ts(n, :, :),rts)
end do
gsn_panel(wks,plot,(/neof,1/),rtsP) ; now draw as one plot
end

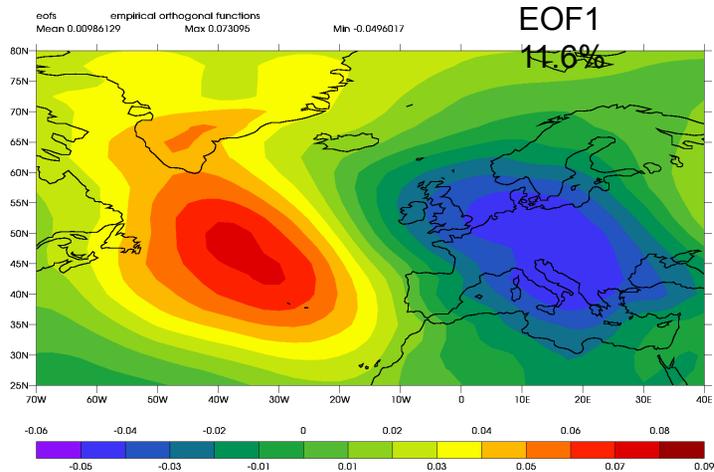
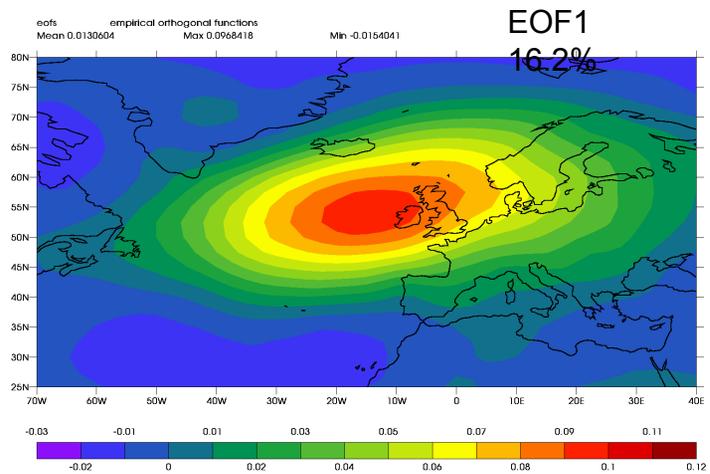
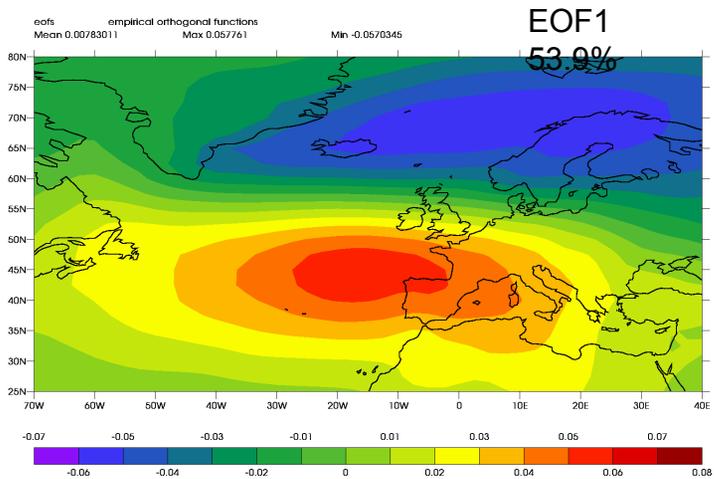
```

# Approaches

- Approach 1 - CDAT + eof2
  - CDAT: Climate Data Analysis Tools
    - <http://www2-pcmdi.llnl.gov/cdat>
    - Separate software subsystems & packages 'glued' under python framework
    - CDAT packages used
      - cdms2 - Climate Data Management System (*file I/O, variables, types, metadata, grids*)
      - cduutil - Climate Data Specific Utilities (*spatial and temporal averages, custom seasons, climatologies*)
      - vcs - Visualization and Control System (*manages graphical window: picture template, graphical methods, data*)
    - Several CDAT installations on discover; `/usr/local/other/SLES11/Cdat/6.0/intel-11.1.069` used for this tutorial
    - Other packages used
      - eof2 - <https://github.com/ajdawson/eof2>
        - » Efficient, even for large data sets
        - » Handles missing values transparently
        - » Automatic meta-data (with cdms2)
        - » Currently installed in `/discover/nobackup/aoloso/GISS/eof2_install`; will be made part of a CDAT installation
        - » Access by adding `/discover/nobackup/aoloso/GISS/eof2_install` to env variable `PYTHONPATH`
      - numpy
      - math
      - pprint

- Approach 1 - CDAT + eof2, cont' d
  - Import necessary modules as listed earlier
  - Few easy steps to compute EOFs
    - `f = cdms2.open('slp.mon.mean.nc')`
      - Open dataset; return dataset object `f`
    - `slp = f('slp', time=('1979','2003'), latitude=(85,20), longitude=(-70, 40))`
      - Power of `cdms2` for sub-setting to create transient variable `slp`
    - `f.close()`
      - Close the dataset
    - `cdutil.setTimeBoundsMonthly(slp)`
      - Put time point at the beginning instead of middle of month (essential)
    - `djfslp = cdutil.DJF(slp)`
      - Extract individual DJF (Dec-Jan-Feb) seasons into variable `djfslp`
    - `slpsolver = Eof(djfslp, weights='coslat')`
      - Create an EOF solver to do the EOF analysis. Square-root of cosine of latitude weights are applied before the computation of EOFs.
    - `eofs = slpsolver.eofs(neofs=3)`
      - Retrieve the three leading EOFs
    - `eigenvalueVec = slpsolver.eigenvalues()`
      - Retrieve the eigenvalues

- Approach 1 - CDAT + eof2, cont' d
  - Looking at the results
    - print eigenvalueVec()
      - Print the eigenvalues
    - print 100\*eigenvalueVec()[0:3]/fsum(eigenvalueVec())
      - Print the percentage contribution of the three leading EOFs
      - [53.9 16.2 11.5]
      - Recollect reference: [53.3 16.5 11.6]
    - p = vcs.init()
      - Initialise a VCS canvas and assign Python variable 'p' to it
    - p.plot(eofs[0], 'default', 'isofill')
      - Plot the first EOF on canvas 'p'



- Approach 1 - CDAT + eof2, cont' d
  - Writing results to a NetCDF file
    - `fout = cdms2.open('eof_results.nc','w')`
      - Open a new file for write
    - `fout.write(eigenvalueVec())`
      - Write the eigenvalues
    - `fout.write(eofs())`
      - Write the three leading EOFs
    - `fout.close()`
      - Close the file

- Approach 1 - CDAT + eof2, cont' d
  - Python code

```
#!/usr/bin/env python
import cdms2, cdutil, vcs
from eof2 import Eof
from pprint import pprint as pp
from numpy import *
from math import *
f = cdms2.open('slp.mon.mean.nc' )
slp = f('slp', time=('1979','2003'), latitude=(25,80),
longitude=(-70, 40))
cdutil.setTimeBoundsMonthly(slp)
djfslp = cdutil.DJF(slp)
slpsolver = Eof(djfslp, weights='coslat')
eigenvalueVec = slpsolver.eigenvalues()
print eigenvalueVec
print eigenvalueVec.shape
eofs = slpsolver.eofs(neofs=3)
eigenSum = fsum(eigenvalueVec())
print eigenSum
p = vcs.init()
p.plot(eofs[0], 'default', 'isofill' )
```

```
#using cdms2 capability to dump EOFs to a NetCDF file

# set options to dump NetCDF3 Classic files
cdms2.setNetcdfDeflateLevelFlag(0)
cdms2.setNetcdfDeflateFlag(0)
cdms2.setNetcdfShuffleFlag(0)

#open a new NetCDF file for “write only”
fout = cdms2.open('eof_results.nc','w')

#write eigenvalues
fout.write(eigenvalueVec())

#write EOFs
fout.write(eofs())

#close file (data not in file until file is closed)
fout.close()
```

# PyClimate

- Python package for analysis of climate variability
- Manipulates netCDF files
- Performs EOF analysis
- Part of SIVO-PyD
- Last release was in 2008!

# Sample EOFs PyClimate Code

```
from numpy import *
from Scientific.IO.NetCDF import *
from pyclimate.svdeofs import *
from pyclimate.ncstruct import *
import math

def deg2rad(d):
    return d*math.pi/180.

inc = NetCDFFile("myFile_with_SLP.nc")
slp = inc.variables["SLP"]
areafactor = sqrt(cos(deg2rad(inc.variables["lat"][:]))).round(7)
slpdata = slp[:, :, :] * areafactor[newaxis, :, newaxis]
oldshape = slpdata.shape
slpdata.shape = (oldshape[0], oldshape[1]*oldshape[2])

pcs, lambdas, eofs = svdeofs(slpdata)
```

